NVIDIA

IMPLEMENTING COMPUTER VISION AND IMAGE PROCESSING SOLUTIONS WITH VPI

Rodolfo Lima, Feb. 11th, 2021

AGENDA
What we will cover today

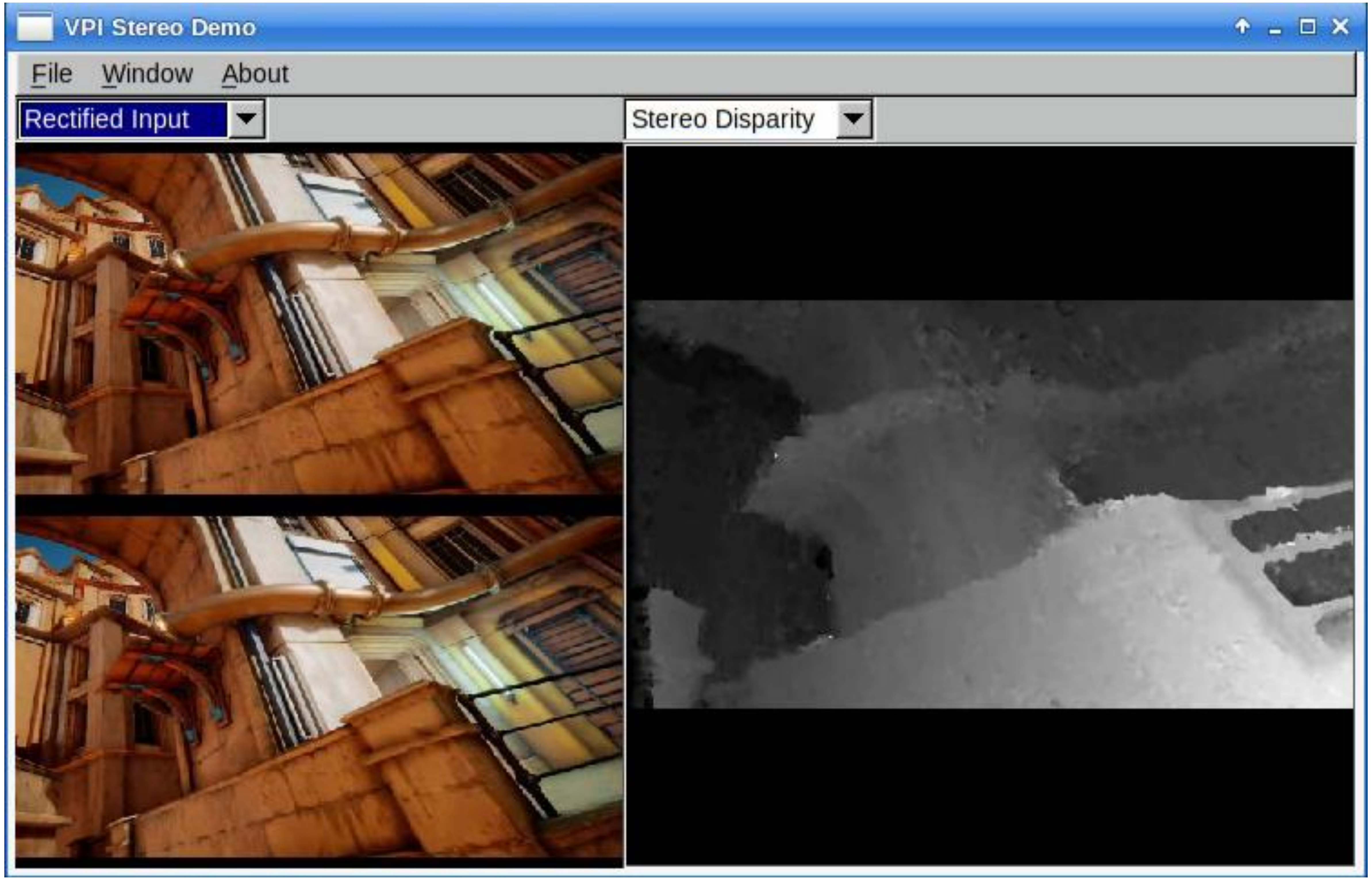
Why VPI?

Core Concepts

Example: Stereo Disparity Estimation

Future Work

Q & A



WHY VPI?

PROBLEM STATEMENT

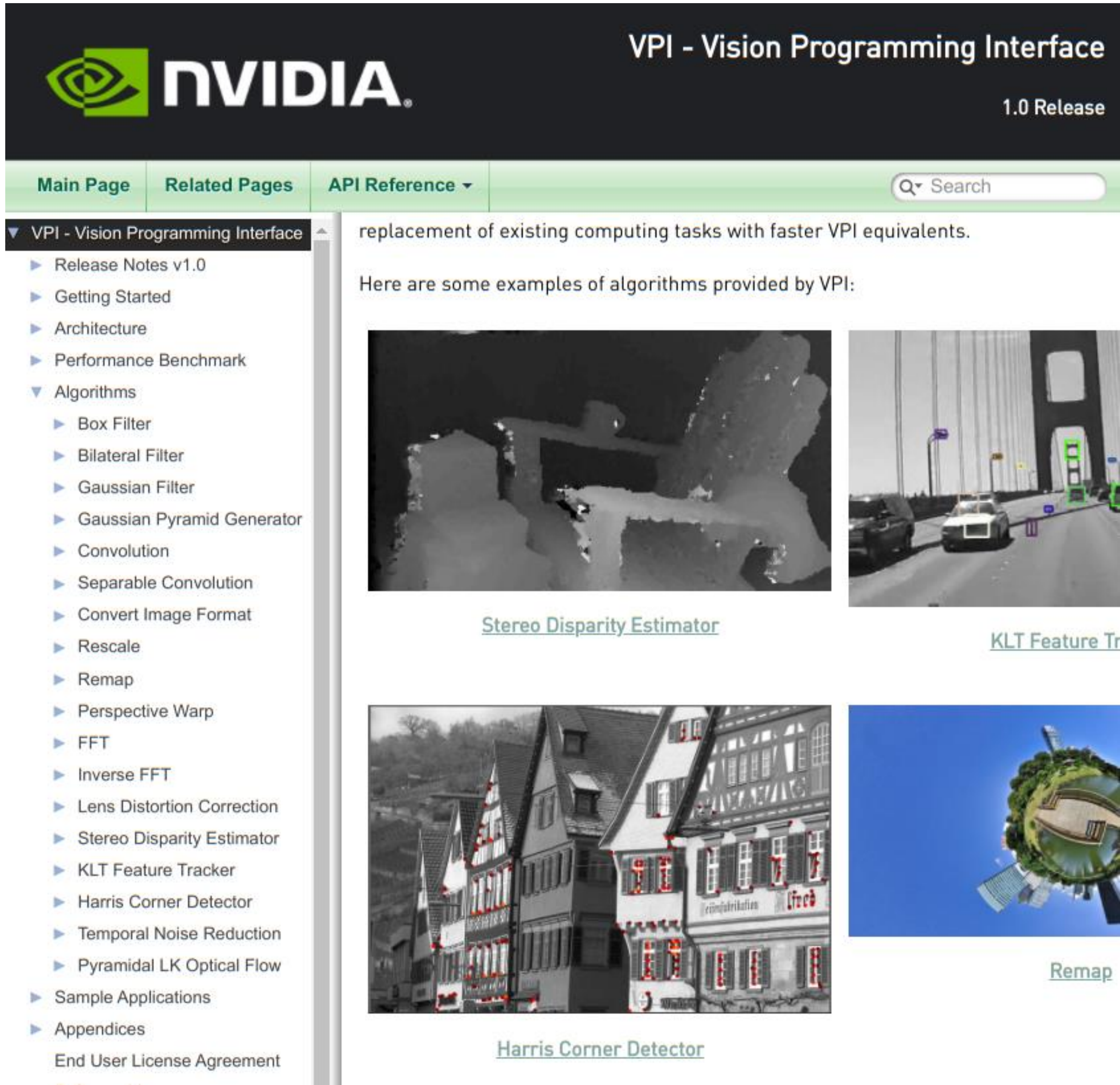
What makes current solutions too complicated?

- Multiple mutually incompatible APIs needed
- Efficient memory management hard to get right
- Difficult to experiment with the code or fine-tune it
- Some compute engines aren't directly accessible via a public API.

VISION PROGRAMMING INTERFACE - VPI

NVIDIA's next-gen API for embedded CV

- Create efficient CV pipelines with all Jetson embedded accelerators
- Same algorithm implemented by different accelerators
- Easily load-balance CV workloads at system level
- Unified API to interface with different accelerators
- Accelerate on both Jetson and x86 Linux PCs
- Zero-copy memory management among different accelerators*
- Interoperability with OpenCV, NVIDIA® CUDA®, EGL, among others
- Designed to supersede NVIDIA® VisionWorks™

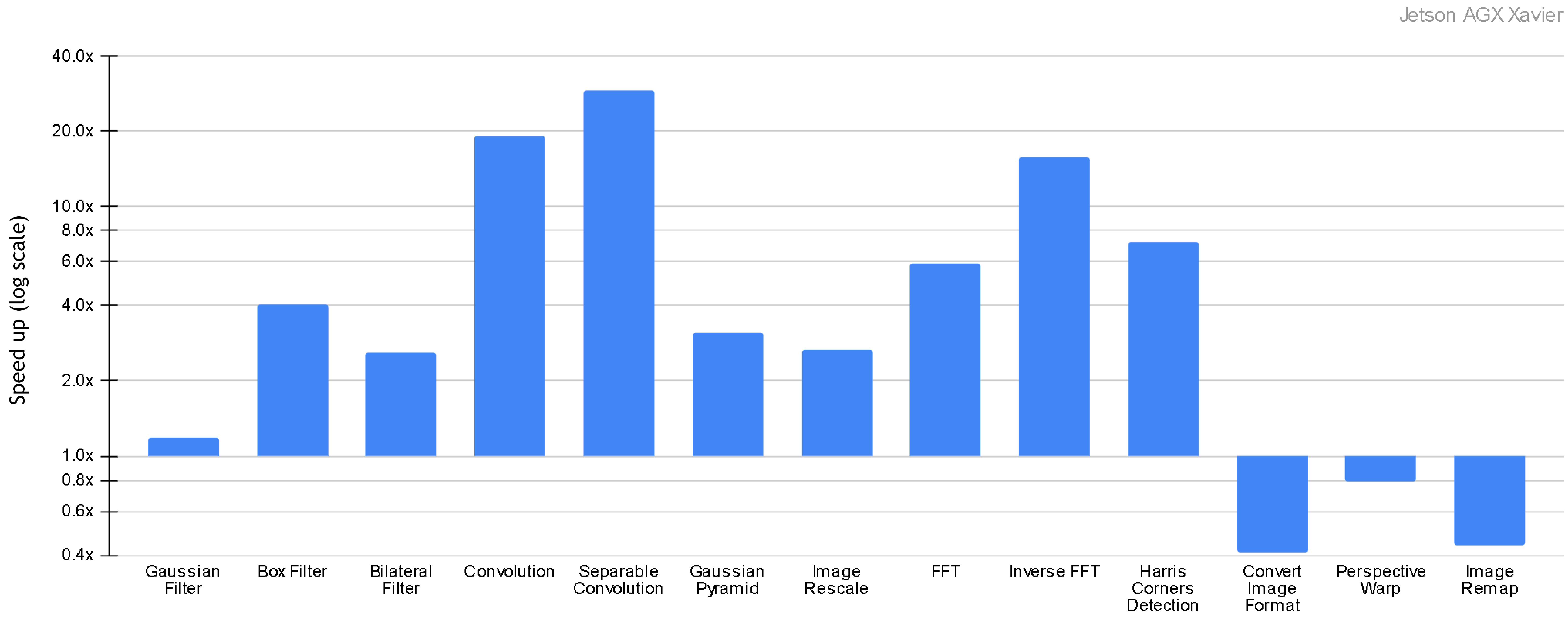


* When allowed by backend and memory characteristics

CPU PERFORMANCE BENCHMARK

Significant Speed Up Compared to OpenCV

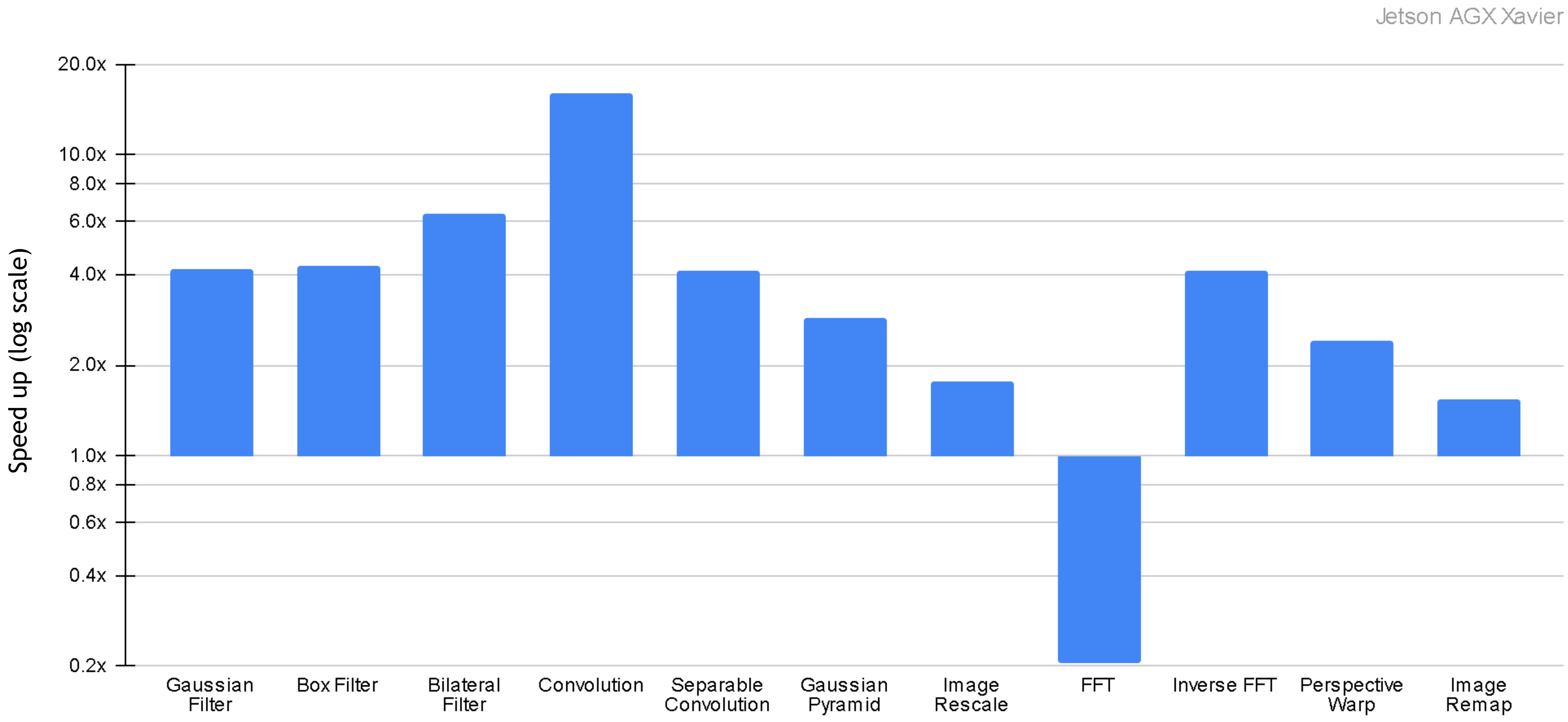
Up to 30x faster than OpenCV on CPU



GPU PERFORMANCE BENCHMARK

Significant Speed Up Compared to OpenCV

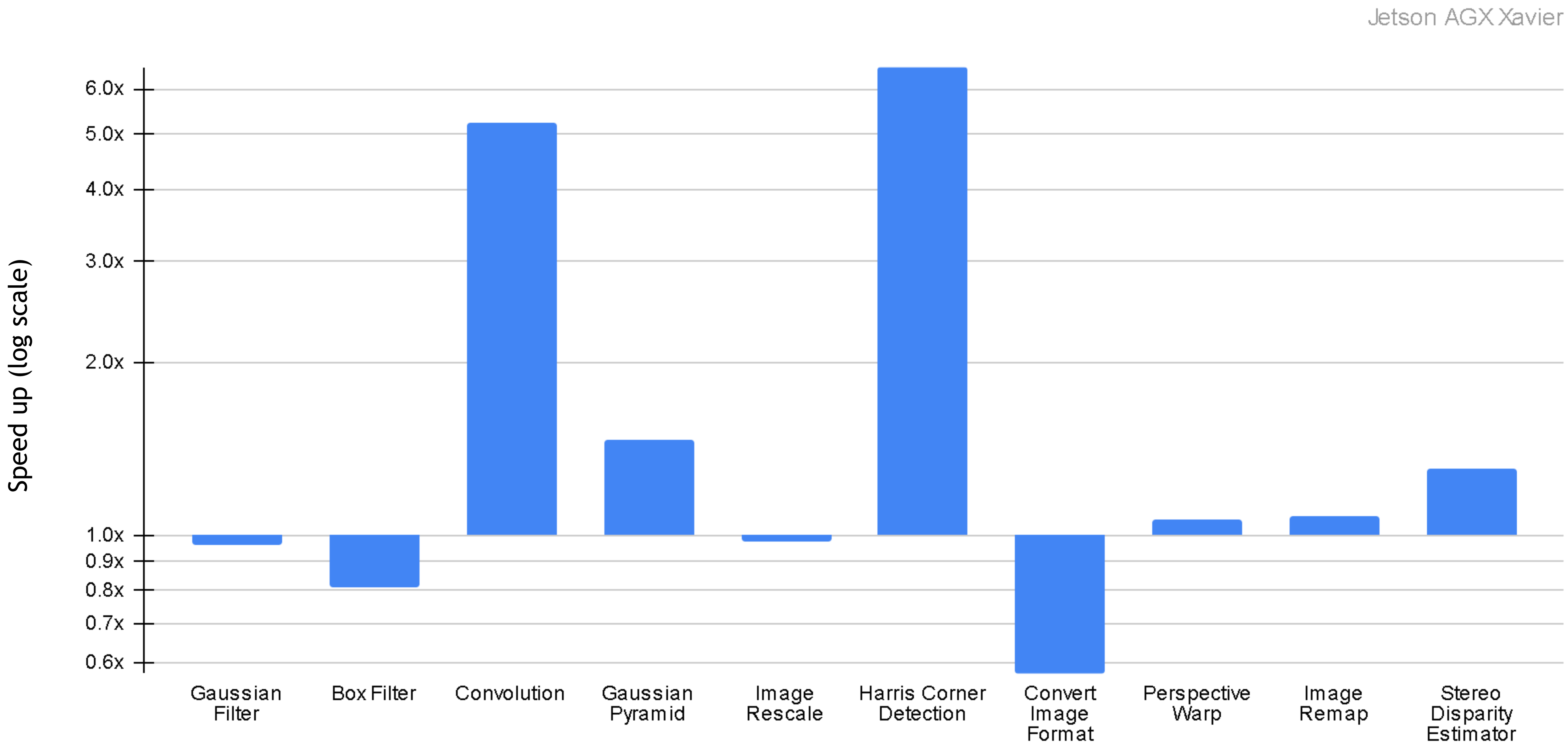
Up to 15x faster than OpenCV on GPU



GPU PERFORMANCE BENCHMARK

Significant Speed Up Compared to VisionWorks

Up to 6x faster than VisionWorks



CORE CONCEPTS

BUILDING BLOCKS

Main VPI components

Backends
CPU
CUDA (GPU)
VIC
PVA

Buffers
Images
Pyramids
Arrays

Algorithms
Gaussian Filter
Stereo Disparity
FFT
...

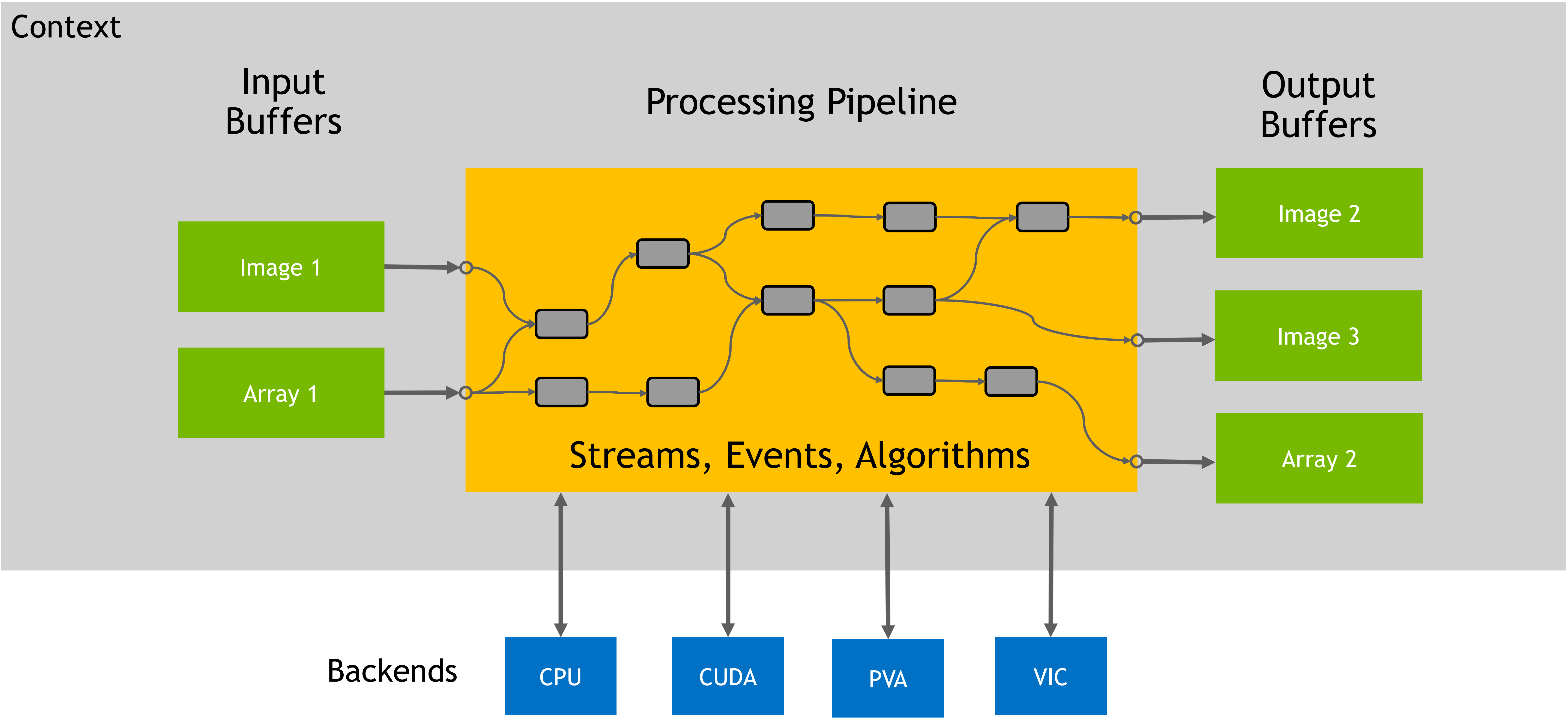
Streams
Execution Queues

Events
Synchronization

Contexts
Resource Management

PROCESSING PIPELINE

How components fit together



BACKENDS

Overview

Backends
CPU
CUDA
PVA
VIC

- A backend represents one type of compute engine
- Certain tasks are more fit to some specific backend
- Same algorithm implemented in multiple backends
    - Allows offloading tasks to less used engines
    - Eg: leave GPU free for DL inference

GPU

VIC

CPU

PVA

Jetson AGX Xavier

GPU

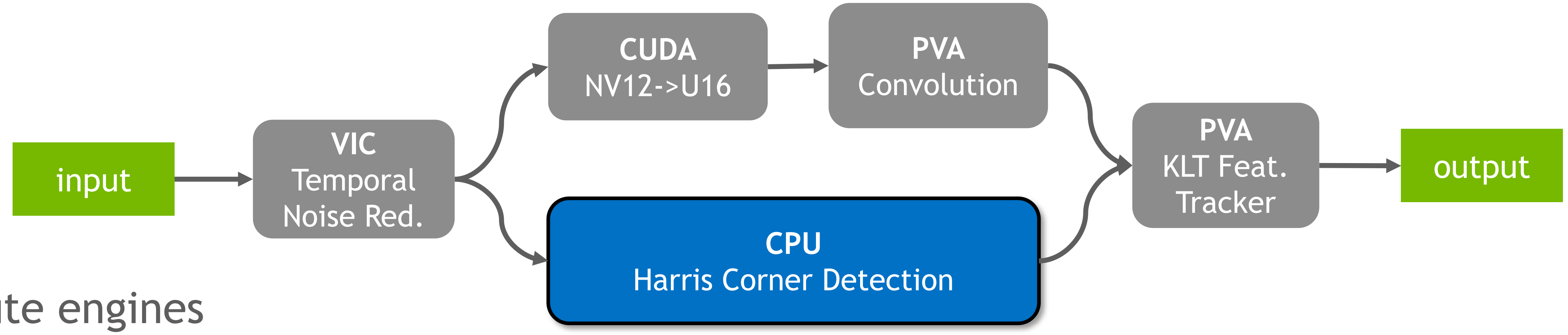
CPU

Geforce RTX 2070 + CPU

BACKENDS

CPU – Central Processing Unit

Backends
CPU
CUDA
PVA
VIC



- Offload computation from other compute engines
- CPU would be left idle otherwise
- More cores -> faster algorithm execution
- Available on Jetson/aarch64 and x86 architectures

BACKENDS

CUDA – Compute Unified Device Architecture

Backends
CPU
CUDA
PVA
VIC

- Represents the GPU on the system
- Speed
- Flexibility – less restrictions on algorithm parameters
- Available on Jetson and x86 architectures
- Supports Maxwell, Pascal, Volta, Turing and Ampere GPU architectures

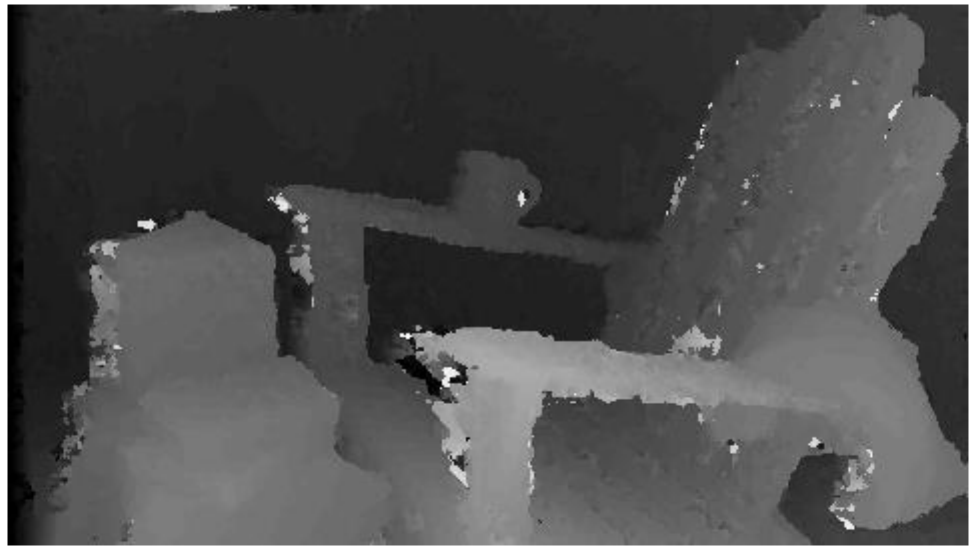
Ampere GA102 GPU HW architecture

BACKENDS

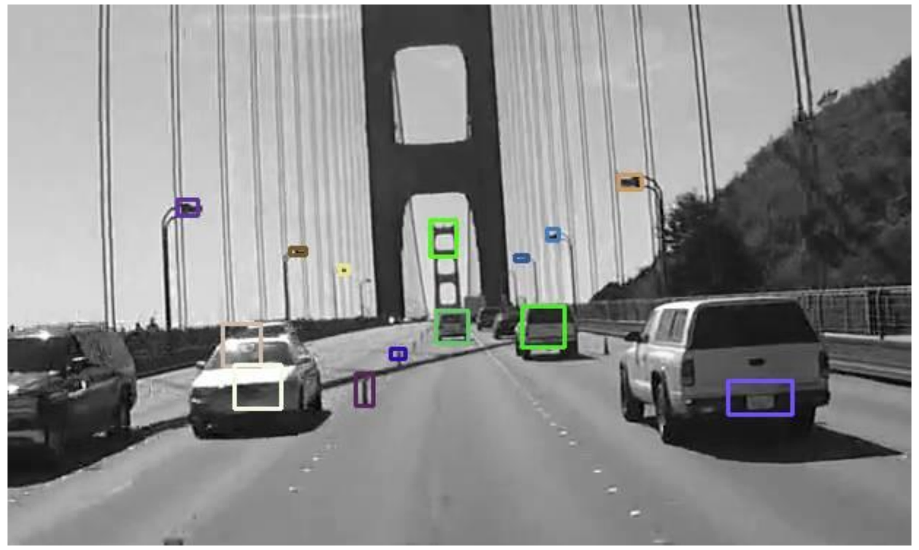
PVA – Programmable Vision Accelerator

Backends
CPU
CUDA
PVA
VIC

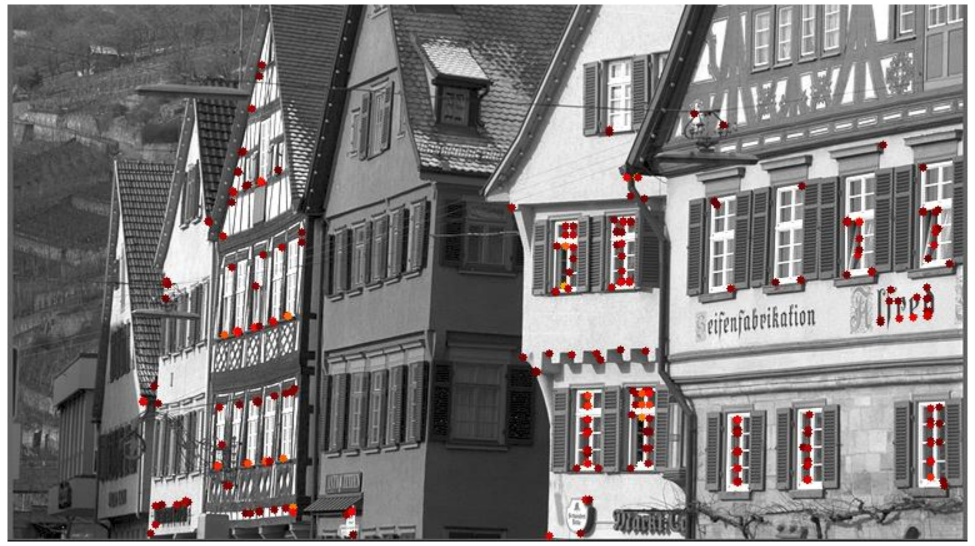
- Used when power efficiency is required
- Available on Jetson AGX Xavier and Jetson Xavier NX devices.
- Four independent processors per device
- Increased task parallelism
- Good choice to offload processing from GPUs.



Stereo Disparity Estimator



KLT Feature Tracker



Harris Corner Detection

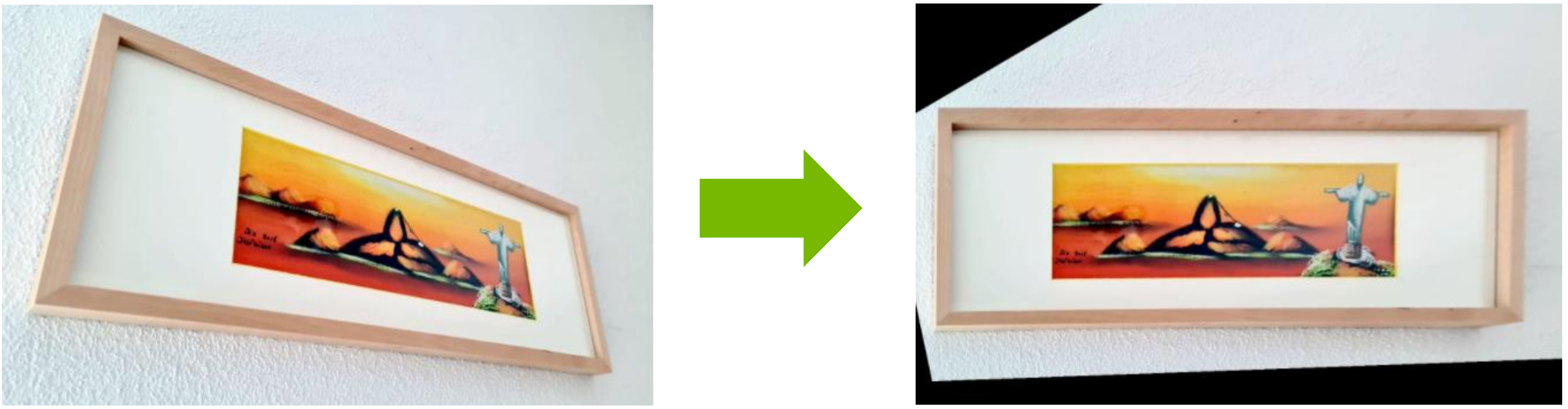
Gaussian Filter

BACKENDS

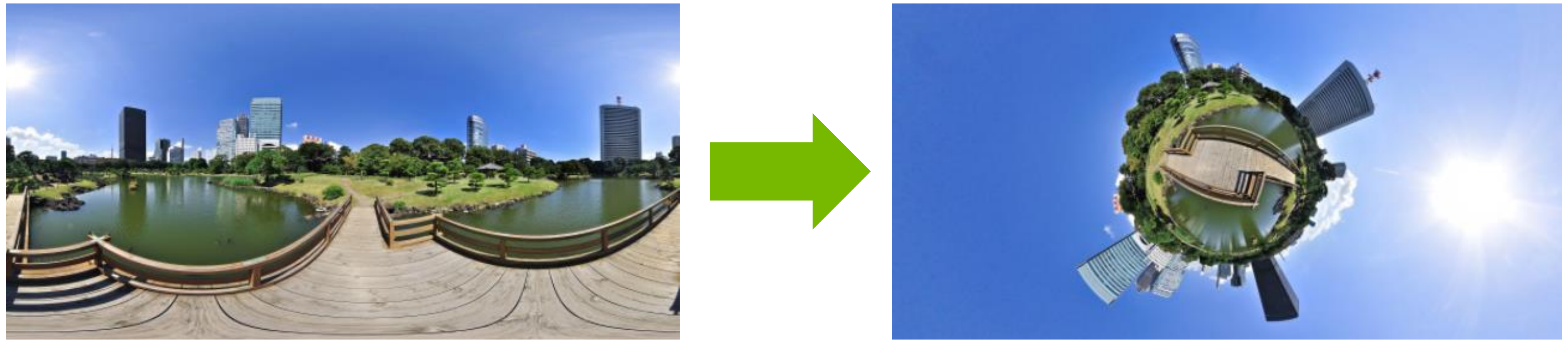
VIC - Video Image Compositor

Backends
CPU
CUDA
PVA
VIC

- Fixed-function processor
- Low power usage
- Optimized for video processing
- Available on all Jetson devices
- Some operations available only on Jetson AGX Xavier and Jetson Xavier NX devices



Perspective Warp



Remap

BUFFERS

VPIImage overview

Buffers
Image
Pyramids
Arrays

- Images are characterized by:
  - Dimensions (width x height)
  - Image Format – NV12, U8, RGB8, …
  - Backends that can use them: CUDA, CPU, PVA, …

BUFFERS

VPIImage usage

Buffers

Image

Pyramids

Arrays

- Images are characterized by:
    - Dimensions (width x height)
    - Image Format – NV12, U8, RGB8, …
    - Backends that can use them: CUDA, CPU, PVA, …

```
   /* Create a 512x256 RGBA8 VPIImage
      to be used by CPU and CUDA backends. */
VPIImage img;
vpiImageCreate(512, 256,
               VPI_IMAGE_FORMAT_RGBA8,
               VPI_BACKEND_CPU|VPI_BACKEND_CUDA,
               &img);

/* (omitted) use img */

   /* When finished, destroy/deallocate it */
vpiImageDestroy(img);
```

BUFFERS

VPIImage – OpenCV Interoperability

Buffers
Image
Pyramids
Arrays

- Images can wrap user-allocated memory buffers:
  - OpenCV
  - CUDA device pointer (cudaMalloc, …)
  - Raw CPU pointer (malloc, new, …)
  - EGLImage*
  - NvBuffer*

```
/* Load an image from disk using OpenCV */
cv::Mat cvImage = cv::imread("baboon.png");

/* Wrap it to be used by VPI.
   Image format inferred by VPI = BGR8 */
VPIImage img;
vpiImageCreateOpenCVMatWrapper(cvImage,
                     VPI_BACKEND_CPU|VPI_BACKEND_CUDA,
                     &img);

/* Set wrapper to another cv::Mat */
vpiImageSetWrappedOpenCVMat(img, cvImage2);

/* When finished, destroy the wrapper */
vpiImageDestroy(img);

/* Now cvImage can be deallocated */
cvImage = cv::Mat();
```

* On Jetson devices only

BUFFERS

VPIImage – CUDA Interoperability

Buffers

Image

Pyramids

Arrays

- Images can wrap user-allocated memory buffers:
  - OpenCV
  - CUDA device pointer (cudaMalloc, …)
  - Raw CPU pointer (malloc, new, …)
  - EGLImage*
  - NvBuffer*

```
/* Allocate a 2D CUDA memory buffer for RGBA8 data */
void *devBuffer;
size_t pitch;
cudaMallocPitch(&devBuffer, &pitch, 512*4, 256);

/* Wrap it to be used by VPI. */
VPIImageData data;
memset(&data, 0, sizeof(data));
data.format = VPI_IMAGE_FORMAT_RGBA8;
data.numPlanes = 1;
data.planes[0].data = devBuffer;
data.planes[0].width = 512;
data.planes[0].height = 256;
data.planes[0].pitchBytes = pitch;
VPIImage img;
vpiImageCreateCUDAMemWrapper(&data,
                             VPI_BACKEND_CPU|VPI_BACKEND_CUDA,
                             &img);

/* Set wrapper to a another CUDA memory buffer */
vpiImageSetWrappedCUDAMem(img, &data2);

/* When finished with it, destroy it */
vpiImageDestroy(img);

/* Now devBuffer can be deallocated */
cudaFree(devBuffer);
```

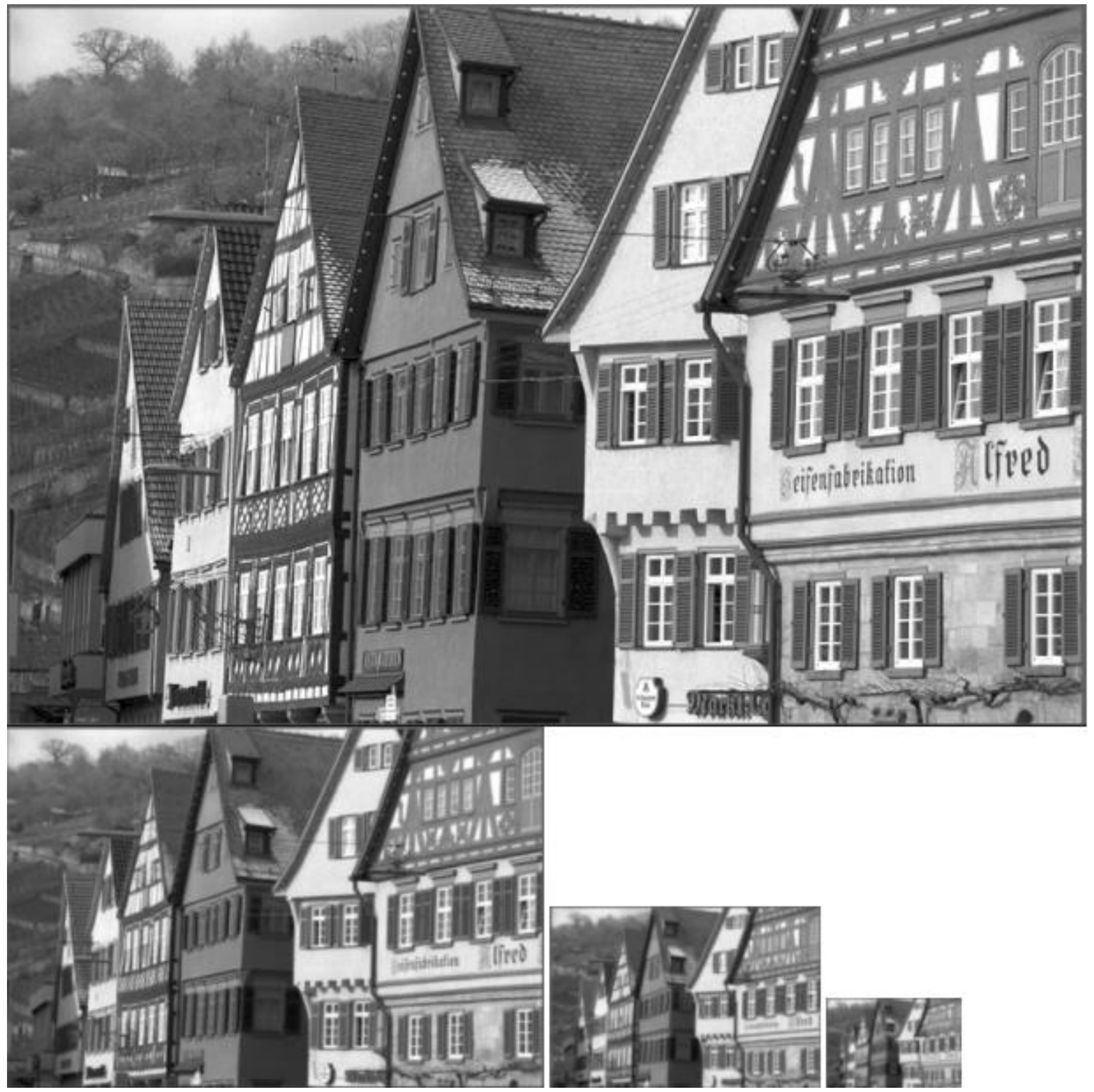
* On Jetson devices only

BUFFERS

VPIPyramid

Buffers
Image
Pyramids
Arrays

- Pyramids are characterized by:
    - Finer level dimensions (width x height)
    - Image Format – NV12, U8, RGB8, …
    - Number of levels
    - Scale between one level and the next
    - Backends that can use them: CUDA, CPU, PVA, …
    - More info: https://docs.nvidia.com/vpi/architecture.html#arch_pyramid



4-level Image Pyramid

BUFFERS

VPIArray

Buffers
Image
Pyramids
Arrays

- Arrays are characterized by:
  - Capacity (maximum number of elements)
  - Array type – key points, scores, bounding boxes …
  - Number of elements (size), which can be redefined after array creation
  - More info: https://docs.nvidia.com/vpi/architecture.html#arch_array

Array of Harris Corners’ (x,y) keypoints

ALGORITHMS

Overview

Algorithms
Gaussian Filter
Stereo Disparity
FFT
...

- Represent computations on buffers
- Same algorithm implemented in several backends
- Different implementations are functionally equivalent
- Easy to switch implementation among available backends

Algorithm	Backend Support			
	CPU	CUDA	PVA[1]	VIC[2]
Box Filter	yes	yes	yes	no
Bilateral Filter	yes	yes	no	no
Gaussian Filter	yes	yes	yes	no
Gaussian Pyramid Generator	yes	yes	yes	no
Convolution	yes	yes	yes	no
Separable Convolution	yes	yes	yes	no
Convert Image Format	yes	yes	no	yes
Rescale	yes	yes	no	yes
Remap	yes	yes	no	yes[1]
Perspective Warp	yes	yes	no	yes[1]
FFT	yes	yes	no	no
Inverse FFT	yes	yes	no	no
Lens Distortion Correction	yes	yes	no	yes[1]
Stereo Disparity Estimator	yes	yes	yes	no
KLT Feature Tracker	yes	yes	yes	no
Harris Corner Detector	yes	yes	yes	no
Temporal Noise Reduction	no	yes	no	yes
Pyramidal LK Optical Flow	yes	yes	no	no

Algorithms available on vpi 1.0

ALGORITHMS

Basic usage

Algorithms

Gaussian Filter
Stereo Disparity
FFT
…

- Represent computations on buffers
- Same algorithm implemented in several backends
- Different implementations are functionally equivalent
- Easy to switch implementation among available backends
- Backend must be enabled in the stream

```
/* Apply a 3x3 Box Filter */
vpiSubmitBoxFilter(stream, VPI_BACKEND_CUDA,
                   imgInput, imgTemp, 3, 3,
                   VPI_BORDER_CLAMP);

/* Rescale the filtered image */
vpiSubmitRescale(stream, VPI_BACKEND_VIC,
                 imgTemp, imgResult,
                 VPI_INTERP_LINEAR,
                 VPI_BORDER_CLAMP, 0);
```

ALGORITHMS

Algorithm payload

Algorithms
Gaussian Filter
Stereo Disparity
FFT
…

- Some algorithms need auxiliary resources
- Such resources are encapsulated in a VPIPayload object
- User is responsible for managing payload lifetime
- Same payload can’t be used concurrently

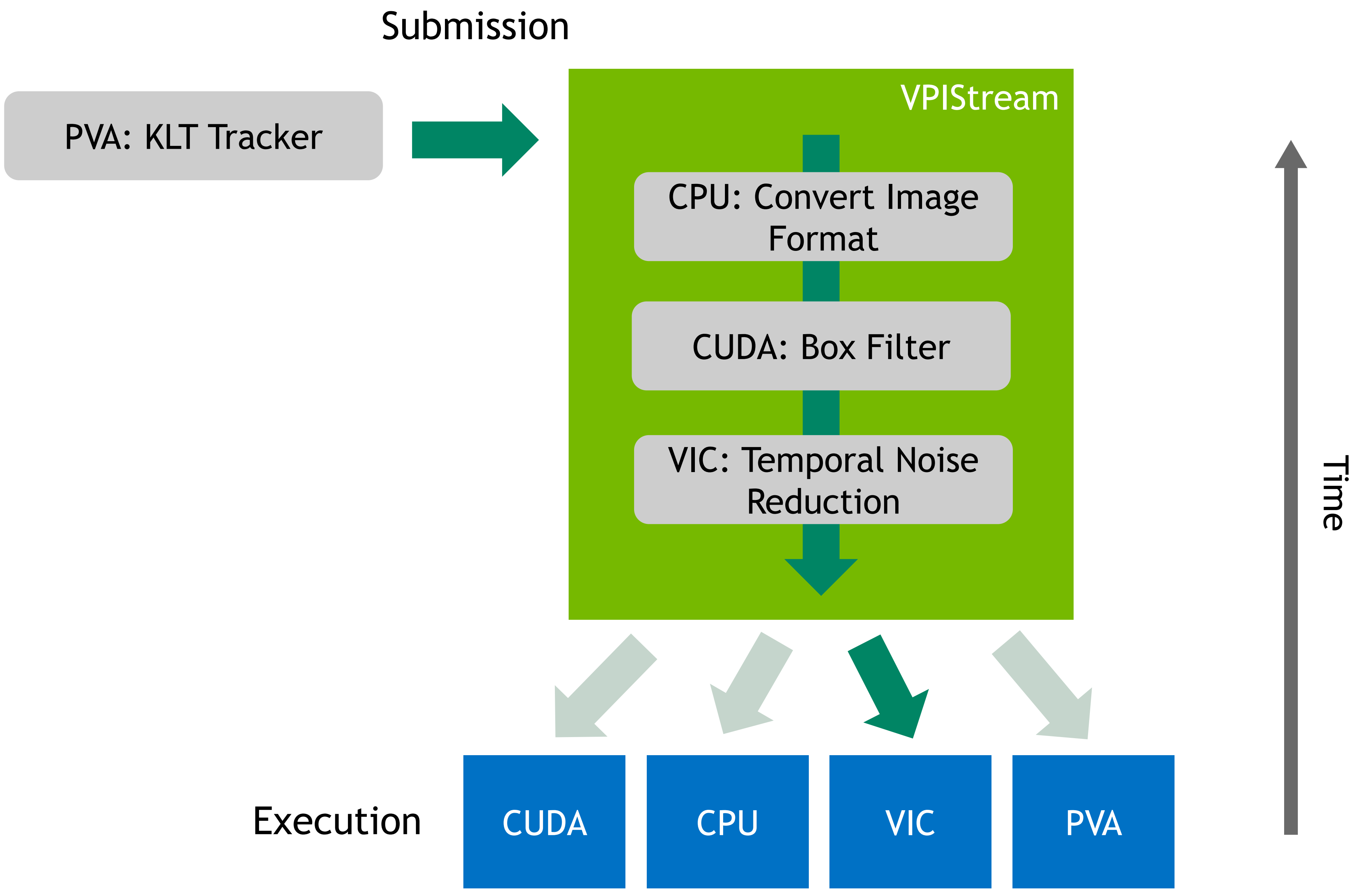
```
/* Create stereo payload */
VPIPayload stereo;
vpiCreateStereoDisparityEstimator(VPI_BACKEND_PVA,
                                  480, 270,
                                  VPI_IMAGE_FORMAT_U16,
                                  NULL, &stereo);

/* Configure some algorithm parameters */
VPIStereoDisparityEstimatorParams params;
params.windowSize   = 5;
Params.maxDisparity = 64;

/* Submits algorithm to process stereo pair */
vpiSubmitStereoDisparityEstimator(stream,
                                  VPI_BACKEND_PVA,
                                  stereo,
                                  imgLeft, imgRight,
                                  outDisparity, NULL,
                                  &params);

/* When not needed anymore, destroy payload */
vpiPayloadDestroy(stereo);
```

STREAMS

Overview

Streams
Execution Queues

- Represents a task execution queue.
- Dispatches algorithms to backends for execution.
- Asynchronous task submission with respect to submission (calling) thread.
- Use of multiple streams allows for parallel task processing.
- Destroying a stream implicitly waits for queued tasks to complete.



STREAMS

VPIStream usage

Streams
Execution Queues

- Represents a task execution queue.
- Dispatches algorithms to backends for execution.
- Asynchronous task submission with respect to submission (calling) thread.
- Use of multiple streams allows for parallel task processing.
- Destroying a stream implicitly waits for queued tasks to complete.

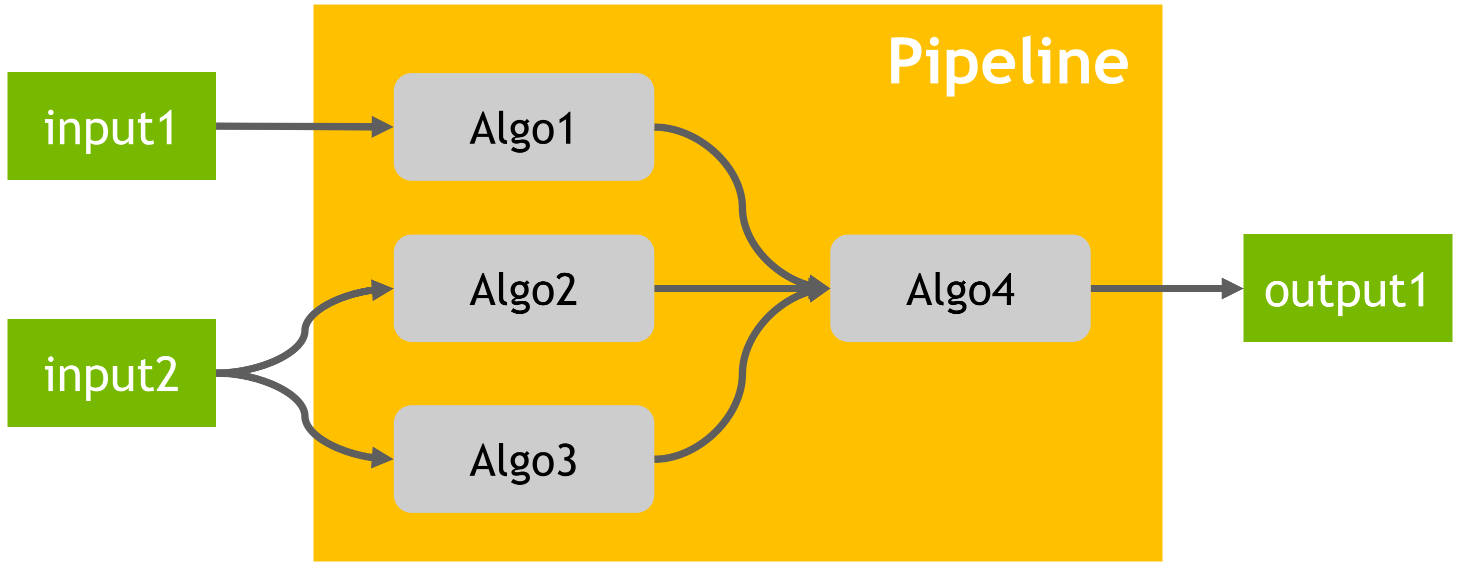
```
/* Create a stream for algorithm execution in
   CUDA, PVA and VIC backends. */
VPIStream stream;
vpiStreamCreate(VPI_BACKEND_CUDA|VPI_BACKEND_PVA|
                VPI_BACKEND_VIC,
               &stream);

/* Apply a 3x3 Box Filter */
vpiSubmitBoxFilter(stream, VPI_BACKEND_CUDA,
                   imgInput, imgTemp, 3, 3,
                   VPI_BORDER_CLAMP);

/* Rescale the filtered image */
vpiSubmitRescale(stream, VPI_BACKEND_VIC,
                 imgTemp, imgResult,
                 VPI_INTERP_LINEAR,
                 VPI_BORDER_CLAMP, 0);

/* Wait until processing ends */
vpiStreamSync(stream);

/* Destroy/deallocate the stream */
vpiStreamDestroy(stream);
```

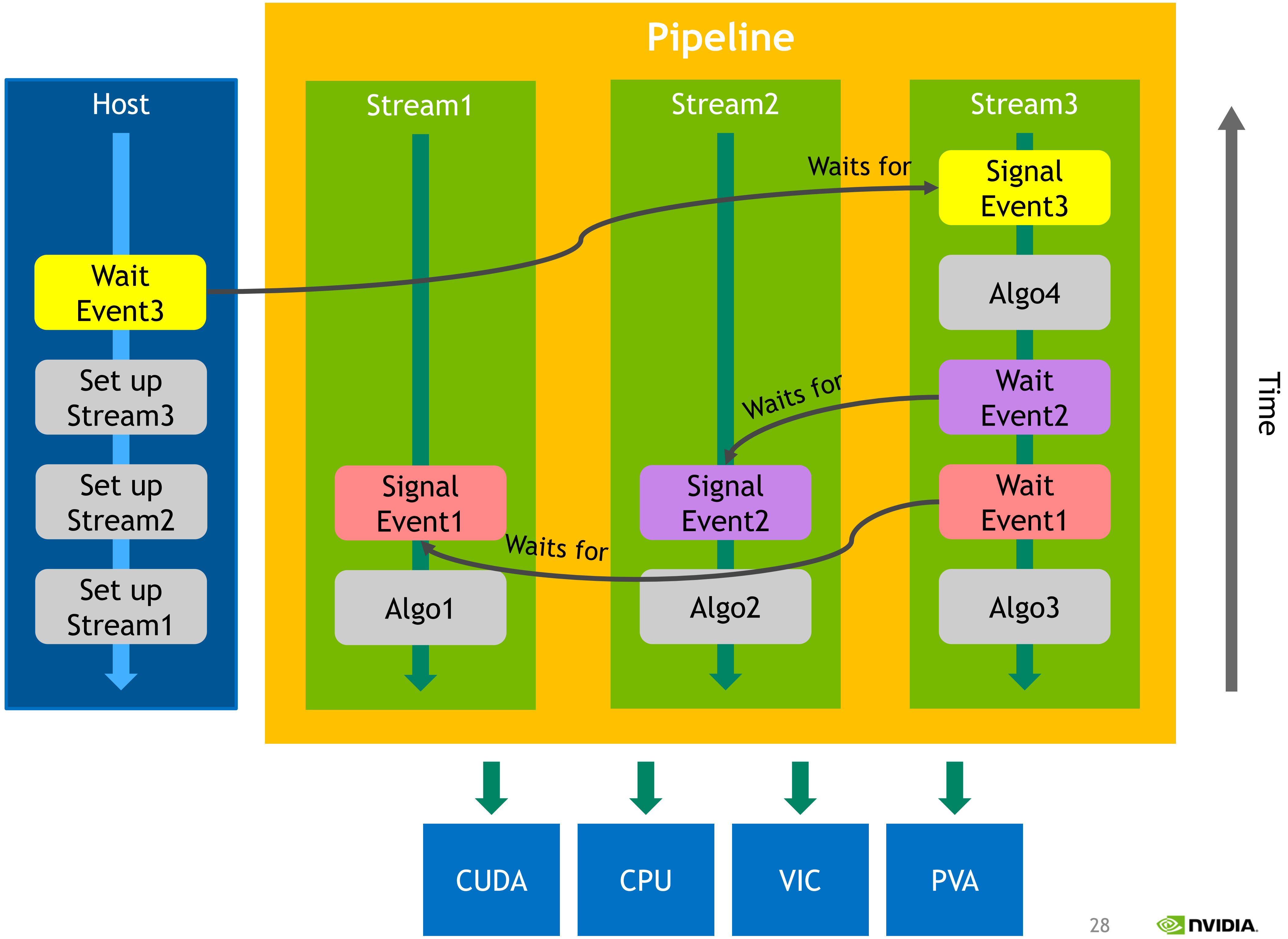
EVENTS

Overview

Events
Synchronization

- Synchronization primitives
- Allow cooperation between streams
- Modeled after NVIDIA CUDA's cudaEvent_t.



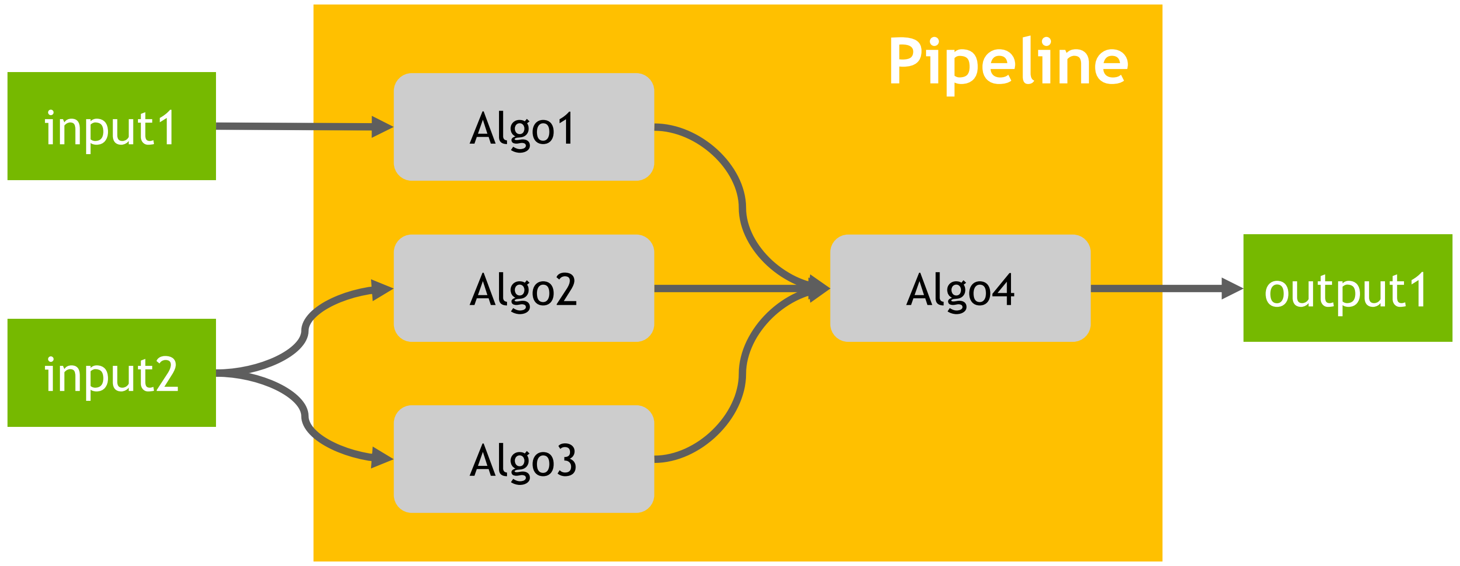


EVENTS

VPIEvent usage

Events
Synchronization

- Synchronization primitives
- Allow cooperation between streams
- Modeled after NVIDIA CUDA's cudaEvent_t




```
/* Create events */
VPIEvent event1, event2;
vpiEventCreate(0, &event1);
vpiEventCreate(0, &event2);

/* Set up stream 1 */
vpiSubmitBoxFilter(stream1, …);          /* Algo1 */
vpiEventRecord(event1, stream1);

/* Set up stream 2 */
vpiSubmitBoxFilter(stream2, …);          /* Algo2 */
vpiEventRecord(event2, stream2);

/* Set up stream 3 */
vpiSubmitConvolution(stream3, …);        /* Algo3 */
vpiStreamWaitEvent(stream3, event1);
vpiStreamWaitEvent(stream3, event2);
vpiSubmitKLTFeatureTracker(stream3, …); /* Algo4 */

/* Wait until processing ends */
vpiStreamSync(stream3);

/* Destroy events when not needed anymore */
vpiEventDestroy(event1);
vpiEventDestroy(event2);
```

ERROR HANDLING

VPIStatus usage

- Most of VPI functions return a VPIStatus
  - Operation succeeded: VPI_SUCCESS
  - Operation failed: VPI_ERROR_*
- vpiGetLastStatusMessage
  - Returns detailed information about last error in current thread
- Asynchronous error reporting
  - Applies to algorithm execution errors
  - Reported by events and new algorithm submissions to stream

```
/* Try to create an image with negative width */
VPIImage img;
VPIStatus status = vpiImageCreate(-1, 256,
                                  VPI_IMAGE_FORMAT_NV12_ER,
                                  0, &img);

/* In case of error, */
if(status != VPI_SUCCESS)
{
   assert(status == vpiPeekLastStatus());

   char buffer[VPI_MAX_STATUS_MESSAGE_LENGTH];
   vpiGetLastStatusMessage(buffer, sizeof(buffer));

   printf(“Error: %s\n  %s\n”, vpiStatusGetName(status),
                               buffer);
   /* prints:
      VPI_ERROR_INVALID_ARGUMENT
        Width must be strictly positive
    */

   exit(1);
}
```

PROGRAM STRUCTURE

How VPI-based programs are structured

1. Initialization

- Allocate resources
  - Images, Arrays, Pyramids
  - Streams, Events
  - Contexts, Payloads
- Pre-compute static/constant data
- Define initial processing state

2. Main Processing

- Fetch input data
- Execute the processing pipeline
- Wait for processing to complete
- Consume result data

3. Cleanup

- Deallocate resources

EXAMPLE:
STEREO DISPARITY ESTIMATION

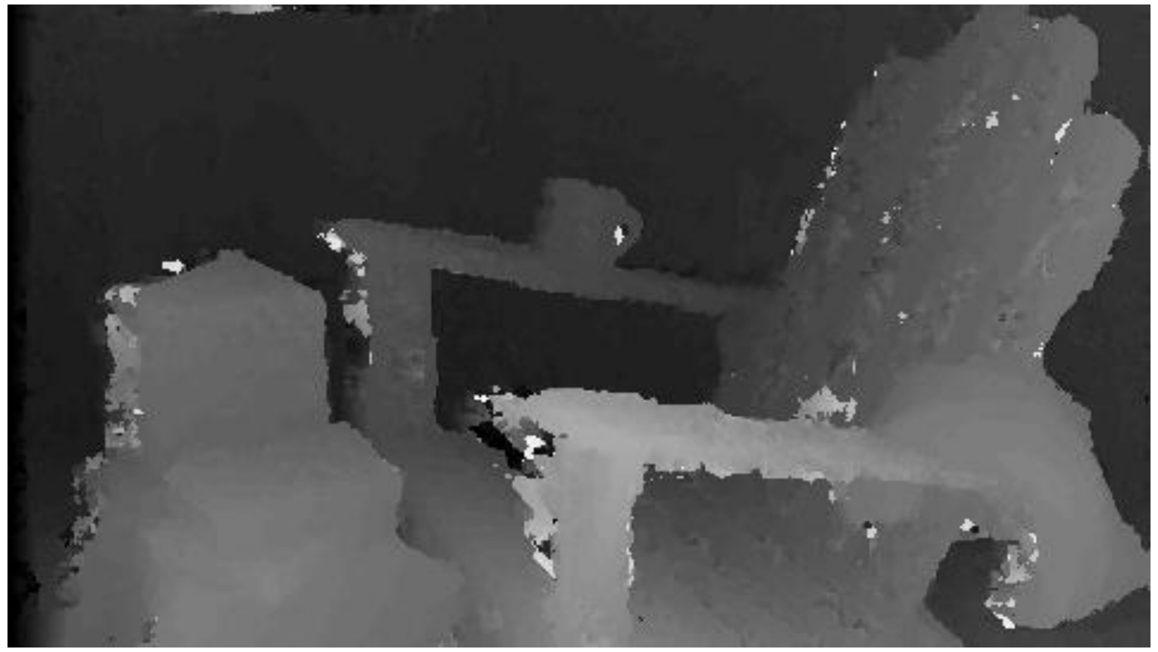
PROBLEM STATEMENT

What is stereo disparity estimation?

- Retrieve depth information from stereo pair
- Stereo pair must be rectified
    - Rows must be aligned
    - Might need Lens Distortion Correction
- Use Semi-Global-Matching technique
    - Match pixels on the left image to pixels on the right
    - Calculate similarity of each pixel in the neighborhood, selects most similar.
    - Horizontal pixel displacement = parallax = stereo disparity
- Achilles' heel
    - Regular patterns/textures
    - Occlusions

Left image

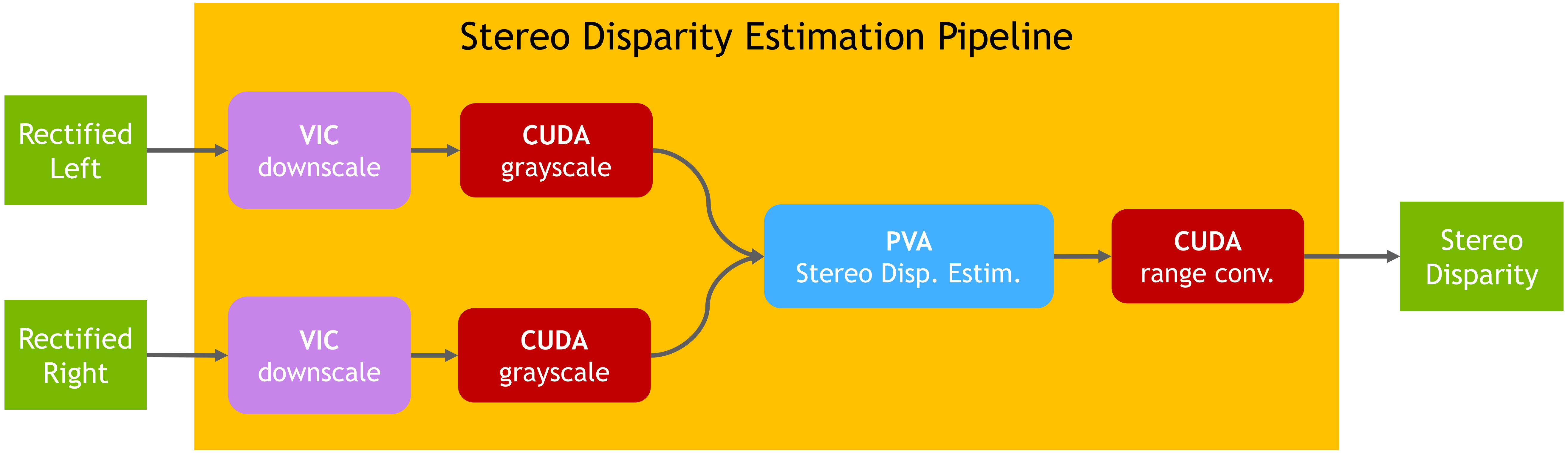
Right image



Estimated stereo disparity

STEREO PIPELINE

Decomposing the stereo pipeline into stages

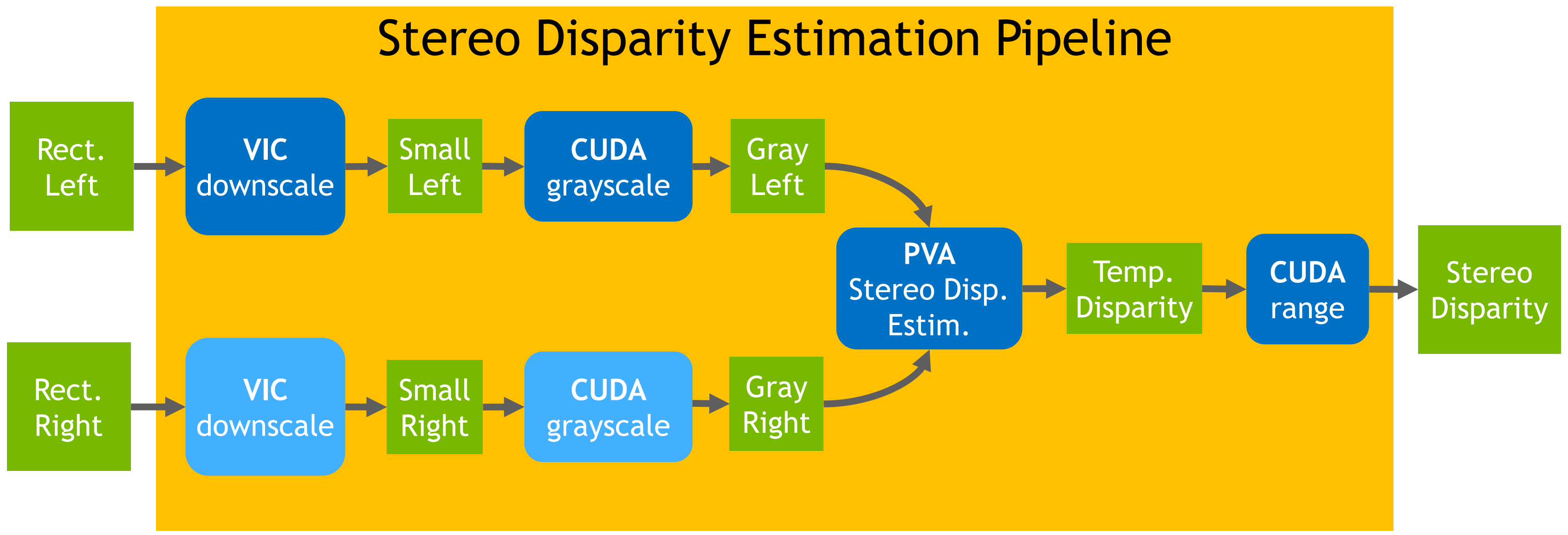


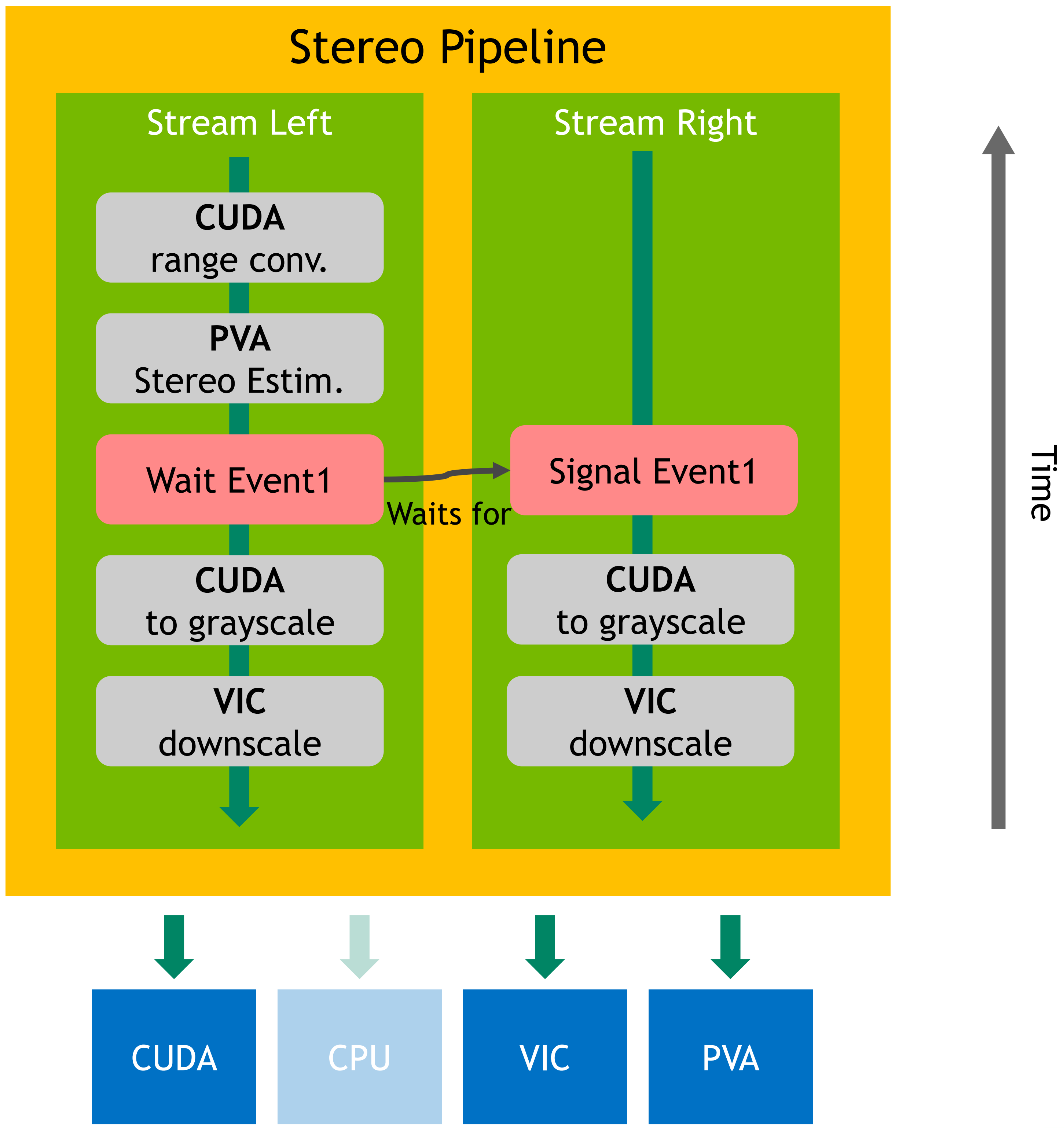
- As simplification, it expects the stereo pair to be rectified already
- Stereo Disparity estimation expects its input to be 480x270, U16 grayscale
- Disparity values' range are rescaled to fit in 0-255 U8 for display
- Different backends work together towards the same goal

STEREO PIPELINE

Decomposing the stereo pipeline into streams

- Stereo pair pre-processing defined as parallel stages
- Reuse left stream for 2nd half of the pipeline
- Resources needed:
  - 2x VPIStream
  - 1x VPIEvent
  - 8x VPIImage: 2x input, 1x output, 5x intermediate
  - 1x VPIPayload for PVA stereo processing

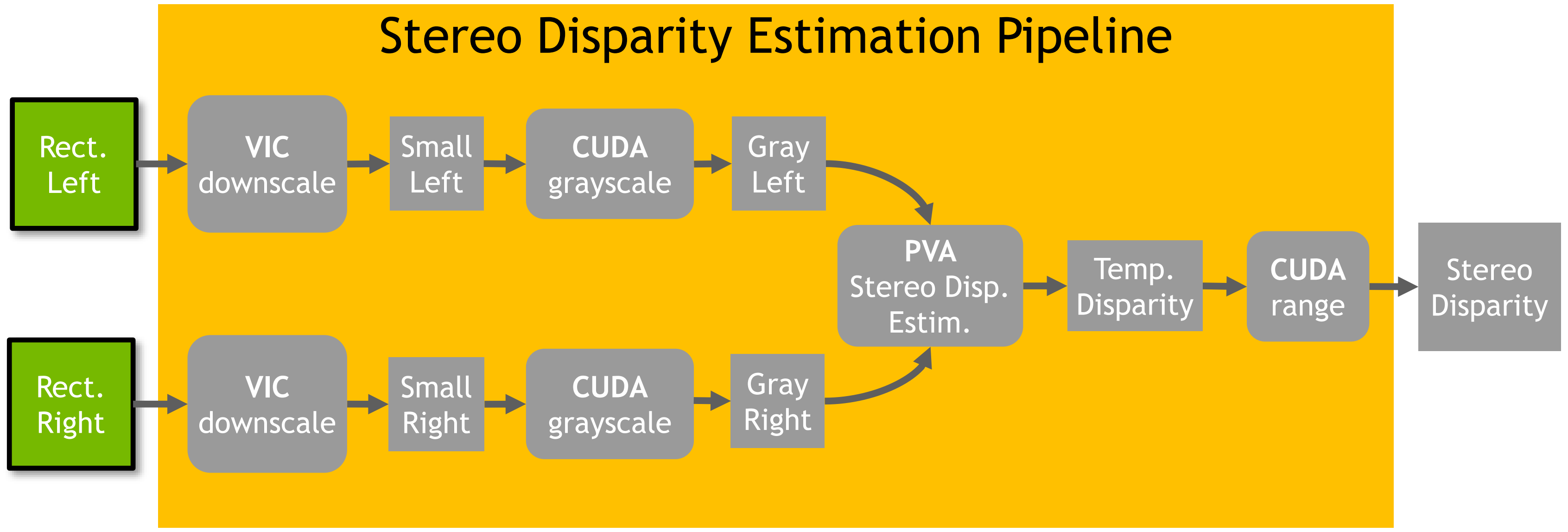




INITIALIZATION PHASE

Allocate input VPIImages

- Stereo pair comes from 2x cv::VideoCapture
- Assuming input format is accepted by VIC downscaler
- Use OpenCV interoperability functions to wrap input cv::Mat into VPIImage
- Input VPIImage object will be re-used to wrap further input stereo pairs in main loop
- Error handling omitted for brevity

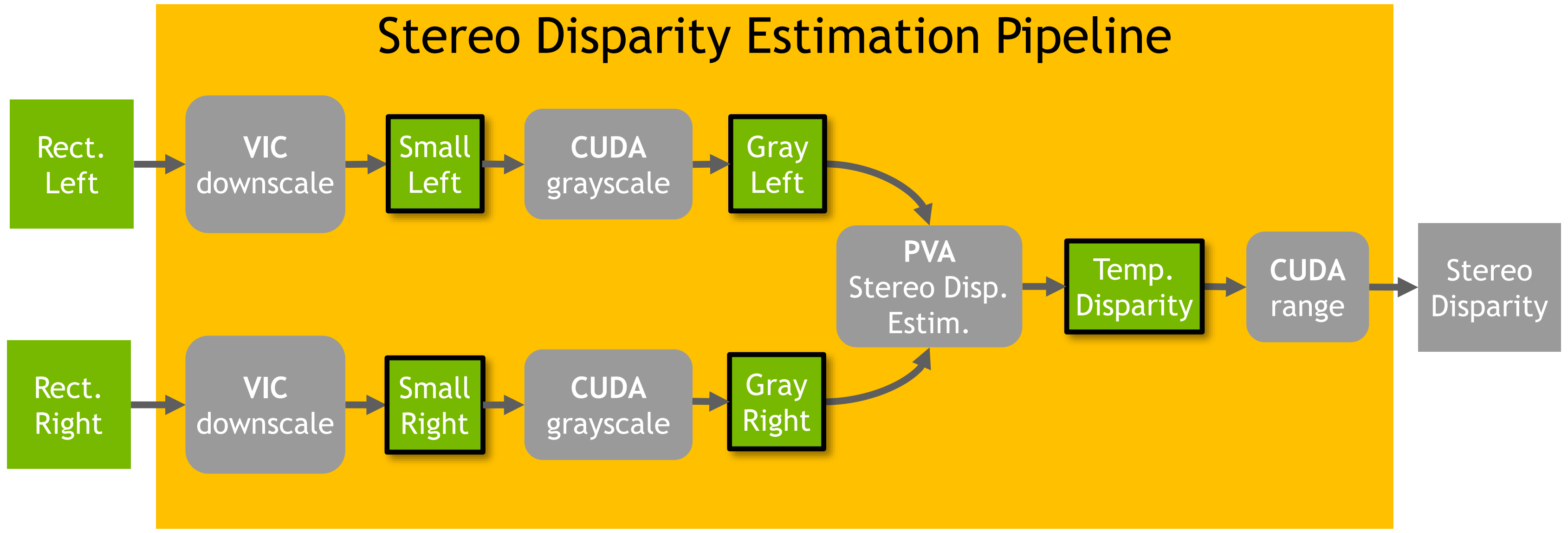



```
/* Include needed headers */
#include <vpi/Image.h>
#include <vpi/Stream.h>
#include <vpi/Event.h>
#include <vpi/OpenCVInterop.hpp>
#include <vpi/CUDAInterop.hpp>
#include <vpi/algo/Rescale.h>
#include <vpi/algo/ConvertImageFormat.h>
#include <vpi/algo/StereoDisparity.h>

/* Fetch first stereo pair using
   OpenCV's cv::VideoCapture */
cv::Mat cvLeft, cvRight;
capLeft.read(cvLeft);
capRight.read(cvRight);

/* Wrap them for VPI usage */
VPIImage inLeft, inRight;

vpiImageCreateOpenCVMatWrapper(cvLeft, VPI_BACKEND_VIC,
                               &inLeft);
vpiImageCreateOpenCVMatWrapper(cvRight, VPI_BACKEND_VIC,
                               &inRight);
```

INITIALIZATION PHASE

Allocate temporary VPIImages

- Intermediateimages must satisfy PVA’s Stereo Disparity Estimation restrictions
  - Input 480x270 U16 (16-bit unsigned values)
  - Output 480x270 Q11.5 – fixed point with 5 bits for fraction = 2^5 = 32 equal unit subdivisions



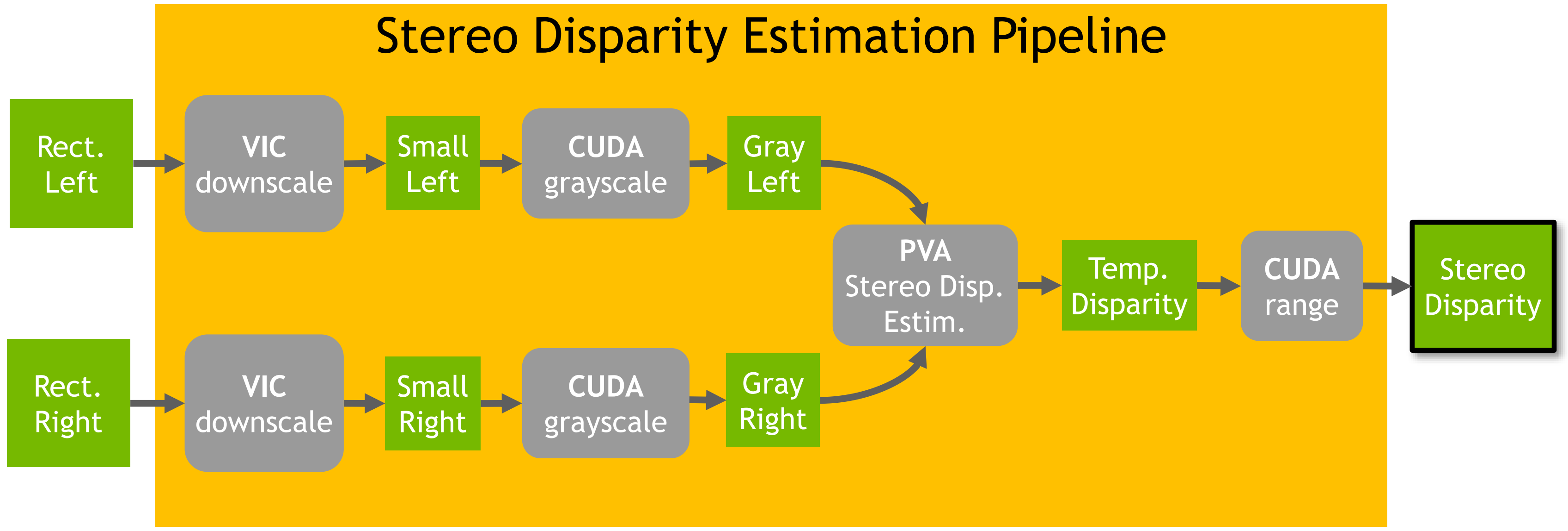

```
/* Get input image format */
VPIImageFormat inFormat;
vpiImageGetFormat(inLeft, &inFormat);

/* Allocate downscaled input images */
VPIImage smallLeft, smallRight;
vpiImageCreate(480, 270, inFormat,
               VPI_BACKEND_VIC|VPI_BACKEND_CUDA,
               &smallLeft);
vpiImageCreate(480, 270, inFormat,
               VPI_BACKEND_VIC|VPI_BACKEND_CUDA,
               &smallRight);

/* Allocate grayscale input images */
VPIImage grayLeft, grayRight;
vpiImageCreate(480, 270, VPI_IMAGE_FORMAT_U16,
               VPI_BACKEND_PVA|VPI_BACKEND_CUDA,
               &grayLeft);
vpiImageCreate(480, 270, VPI_IMAGE_FORMAT_U16,
               VPI_BACKEND_PVA|VPI_BACKEND_CUDA,
               &grayRight);

/* Allocate disparity output */
VPIImage tempDisparity;
vpiImageCreate(480, 270, VPI_IMAGE_FORMAT_U16,
               VPI_BACKEND_PVA|VPI_BACKEND_CUDA,
               &tempDisparity);
```

INITIALIZATION PHASE

Allocate output VPIImage

- Access OpenGL's pixel buffer object (PBO) through CUDA's OpenGL interoperability functions.
- PBO initialization omitted for brevity
- Wrap resulting CUDA memory buffer into a VPIImage

```
/* Create Pixel Buffer Object (PBO) */
GLint pbo;
glGenBuffers(1, &pbo);
/* (Omitted) init `pbo` as a 480x270 8-bit unsigned buffer */

/* Register and map PBO to be accessed by CUDA */
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
cudaGraphicsResource_t gres;
cudaGraphicsGLRegisterBuffer(&gres, pbo, cudaGraphicsMapFlagsNone);
cudaGraphicsMapResources(1, &gres);

/* Retrieve CUDA pointer to PBO memory buffer */
void *ptr;
size_t len;
cudaGraphicsResourceGetMappedPointer(&ptr, &len, gres);

/* Wrap it in a VPIImage */
VPIImageData imgData;
memset(&imgData, 0, sizeof(imgData));
imgData.format = VPI_IMAGE_FORMAT_U8;
imgData.numPlanes = 1;
imgData.planes[0].width = 480;
imgData.planes[0].height = 270;
imgData.planes[0].pitchBytes = len/270;
imgData.planes[0].data = ptr;

VPIImage outDisparity;
vpiImageCreateCUDAMemWrapper(&imgData, VPI_BACKEND_CUDA,
                              &outDisparity);
```

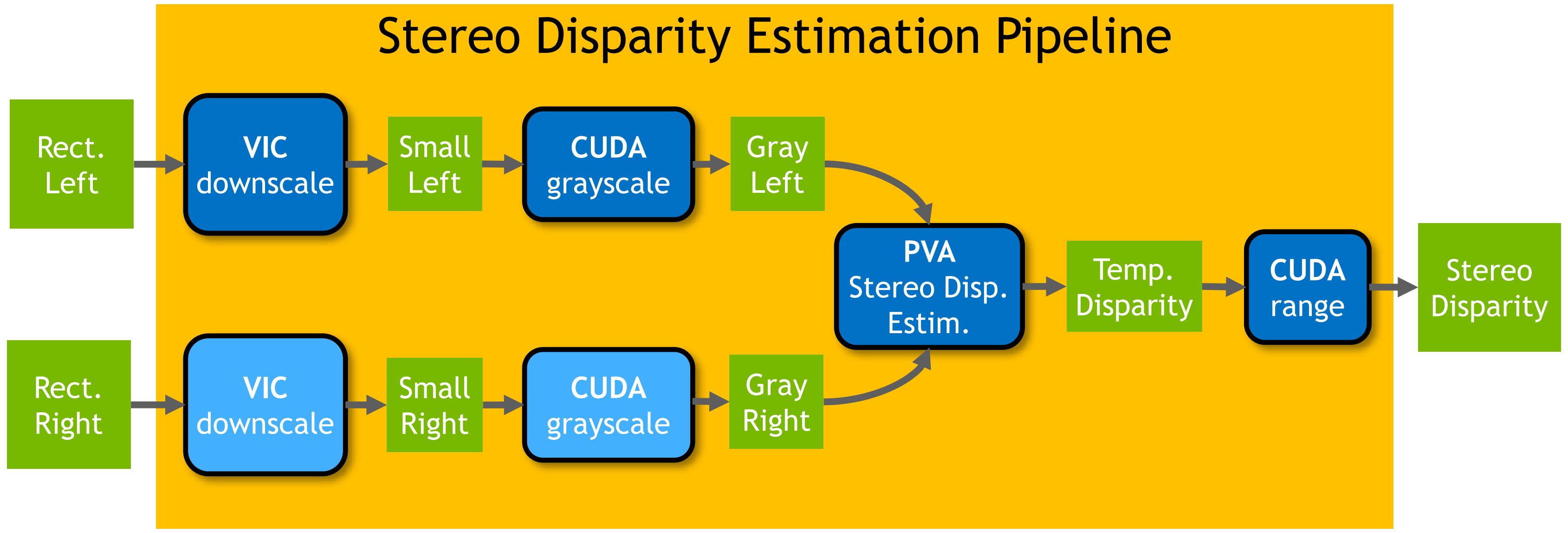


INITIALIZATION PHASE

Allocate VPIStreams, VPIEvent and Stereo VPIPayload

- Creates Stereo Disparity Estimator payload
    - Use maxDisparity = 64 for matching search (default)
- Creates streams with the needed backends enabled
- Created event object for synchronization
    - flags = 0, enabled on all backends for simplicity

```
/* Create stereo payload */
VPIPayload stereo;
vpiCreateStereoDisparityEstimator(VPI_BACKEND_PVA,
                                  480, 270,
                                  VPI_IMAGE_FORMAT_U16,
                                  NULL, &stereo);

/* Create left and right streams */
VPIStream streamLeft, streamRight;
vpiStreamCreate(
      VPI_BACKEND_VIC|VPI_BACKEND_CUDA|VPI_BACKEND_PVA,
      &streamLeft);

vpiStreamCreate(VPI_BACKEND_VIC|VPI_BACKEND_CUDA,
                &streamRight);

/* Create event to be used in stream synchronization */
VPIEvent evSync;
vpiEventCreate(0, &evSync);
```

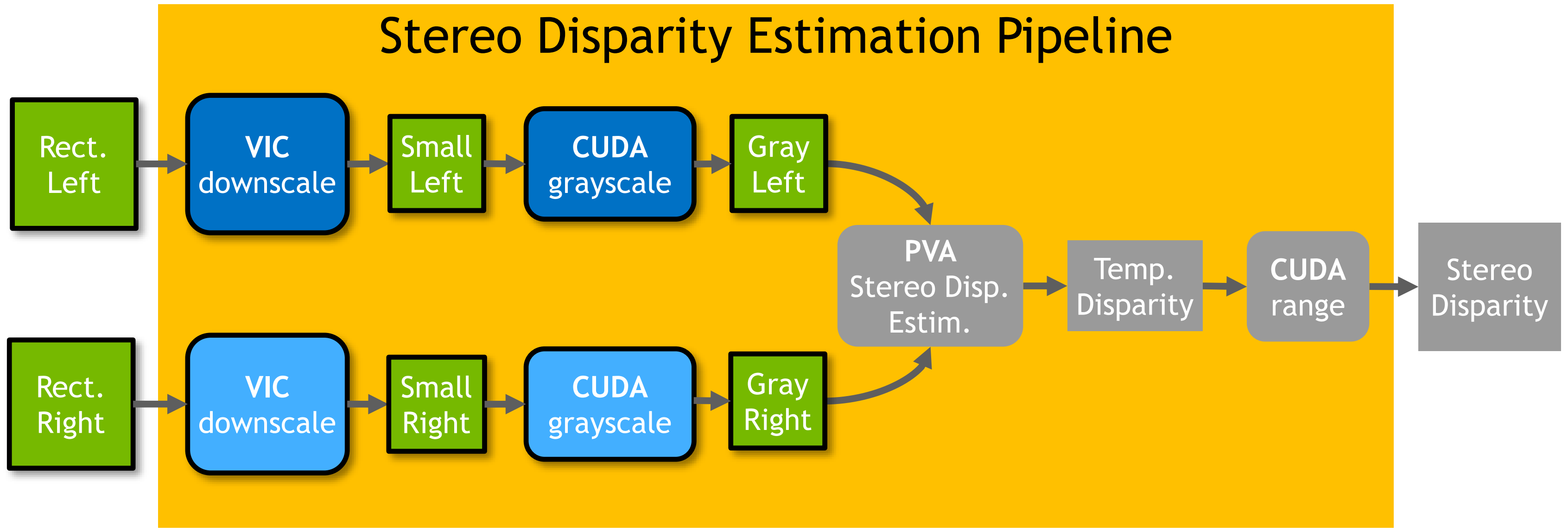


MAIN PROCESSING PHASE

Stereo pair pre-processing

- Retrieve captured stereo pair
- Wrap them in the existing VPIImages: inLeft/inRight
- Submit the pre-processing algorithms to the respective streams

```
/* Retrieve captured stereo pair */
capLeft.read(cvLeft);
capRight.read(cvRight);

/* Wraps them in the existing inLeft/inRight */
vpiImageSetWrappedOpenCVMat(inLeft, cvLeft);
vpiImageSetWrappedOpenCVMat(inRight, cvRight);

/* Left image pre-processing */
vpiSubmitRescale(streamLeft, VPI_BACKEND_VIC,
                 inLeft, smallLeft,
                 VPI_INTERP_LINEAR, VPI_BORDER_CLAMP, 0);

vpiSubmitConvertImageFormat(streamLeft, VPI_BACKEND_CUDA,
                            smallLeft, grayLeft, NULL);

/* Right image pre-processing */
vpiSubmitRescale(streamRight, VPI_BACKEND_VIC,
                 inRight, smallRight,
                 VPI_INTERP_LINEAR, VPI_BORDER_CLAMP, 0);

vpiSubmitConvertImageFormat(streamRight, VPI_BACKEND_CUDA,
                            smallRight, grayRight, NULL);
```

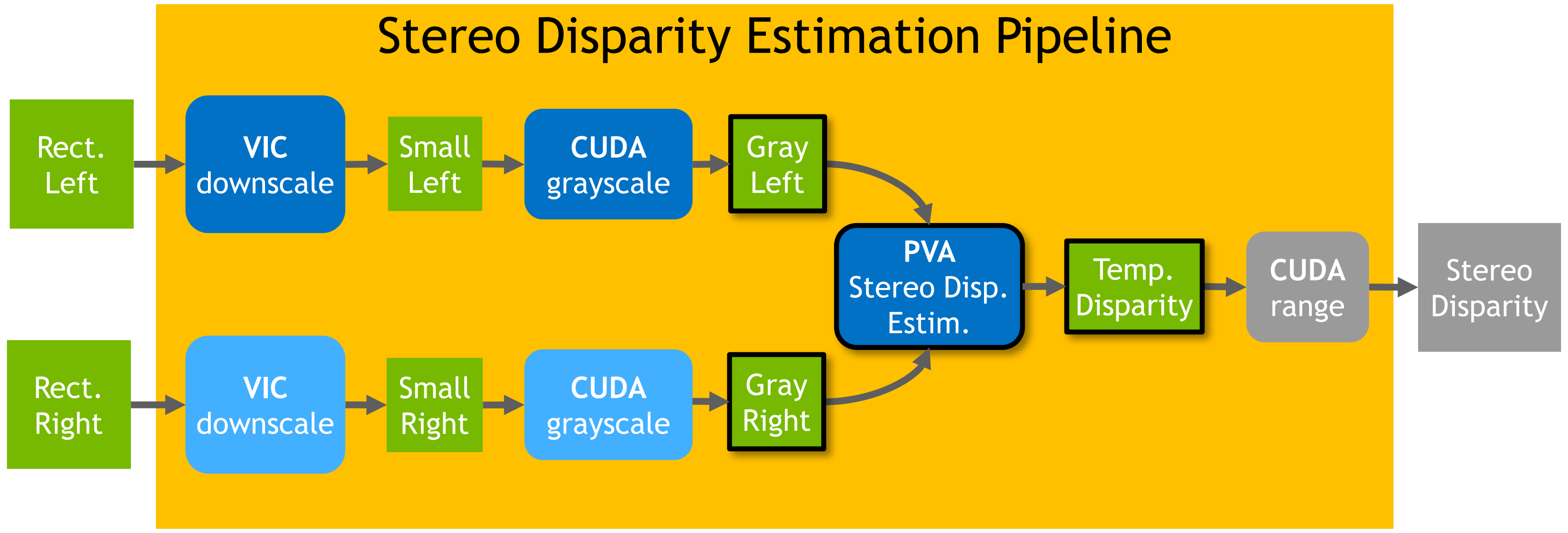


MAIN PROCESSING PHASE

Stereo Disparity Estimation

- Synchronize streams so that pre-processing tasks are guaranteed to be finished
- Submit Stereo Disparity Estimation algorithm to stream
  - Search window size = 5
  - Maximum disparity for matching search = 64

```
/* Record `streamRight` state on evSync */
vpiEventRecord(evSync, streamRight);

/* `streamLeft` will wait until pre-processing on
   `streamRight` is finished -> event is signaled */
vpiStreamWaitEvent(streamLeft, evSync);

/* Configure some algorithm parameters */
VPIStereoDisparityEstimatorParams stereoParams;
stereoParams.windowSize    = 5;
stereoParams.maxDisparity = 64;

/* Submits algorithm to process stereo pair */
vpiSubmitStereoDisparityEstimator(streamLeft,
                                  VPI_BACKEND_PVA,
                                  stereo,
                                  grayLeft, grayRight,
                                  tempDisparity, NULL,
                                  &params);
```

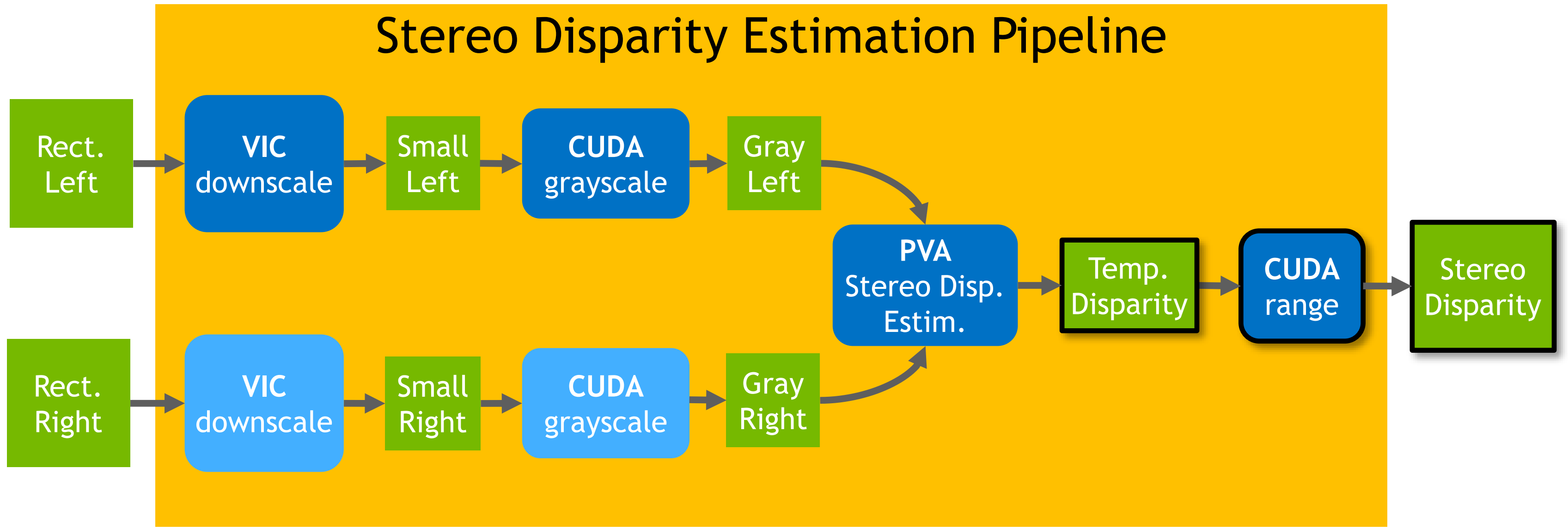


MAIN PROCESSING PHASE

Disparity post-processing

- Convert disparity from U16 (Q11.5) to U8 using CUDA
- Wait until processing is finished
    - Since submit calls are asynchronous, host application can do some other tasks while stream processing isn't finished.
- Display the resulting PBO (omitted)

```
/* Initialize `cvtParams` with default values */
VPIConvertImageFormatParams cvtParams;
vpiInitConvertImageFormatParams(&cvtParams);

/* Scale disparity range from [0-2047] to [0-255] */
cvtParams.scale = 255.0f/(64*32-1);

/* Submit format conversion from U16 (Q11.5) to U8 */
vpiSubmitConvertImageFormat(streamLeft, VPI_BACKEND_CUDA,
                            tmpDisparity, outDisparity,
                            &cvtParams);

/* (Omitted) Host application can do other tasks while
   stream is processing, as submit calls are
   asynchronous */

/* Wait until processing is finished */
vpiStreamSync(streamLeft);

/* (Omitted) Now PBO can be displayed */
```

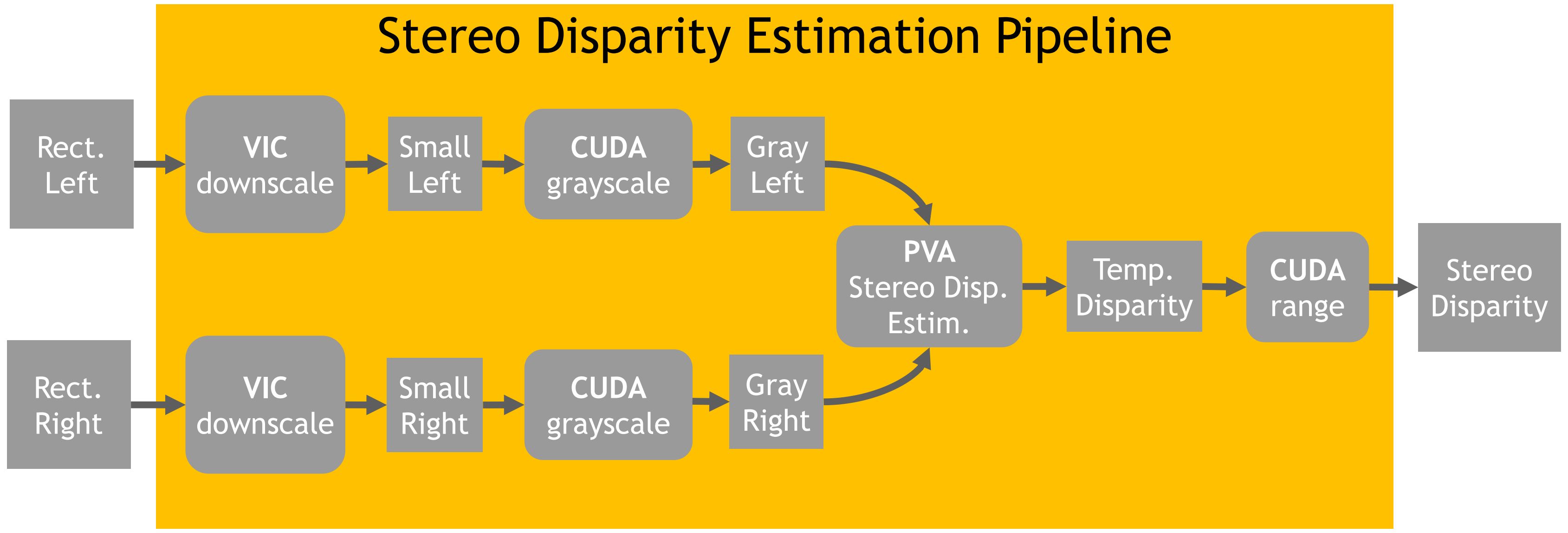


CLEANUP PHASE

Deallocate all resources allocated

- Prefer to destroy streams first
  - Destruction implicitly synchronizes it
  - Guarantees that no tasks are left to be executed
  - Guarantees that no other objects are being or will be used.
- Destroy VPIImage wrappers before wrapped buffers

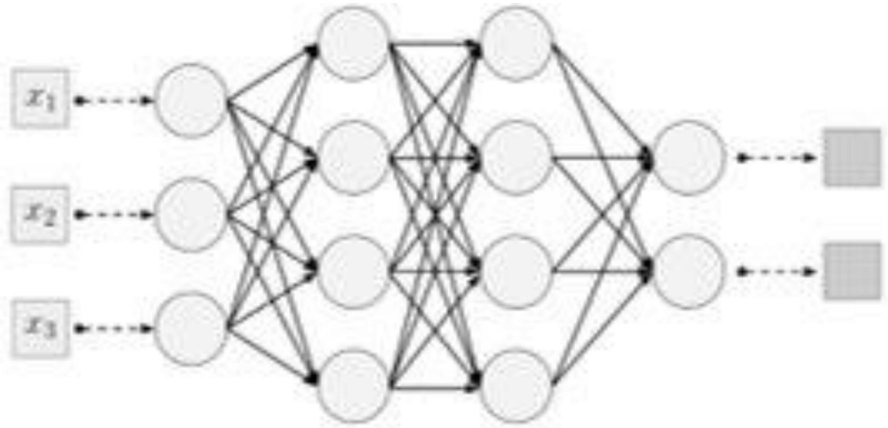
```
/* Deallocate streams first */
vpiStreamDestroy(leftStream); vpiStreamDestroy(rightStream);

/* Deallocate event */
vpiEventDestroy(evSync);

/* Deallocate payload */
vpiPayloadDestroy(stereo);

/* Deallocate images */
vpiImageDestroy(inLeft);     vpiImageDestroy(inRight);
vpiImageDestroy(smallLeft); vpiImageDestroy(smallRight);
vpiImageDestroy(grayLeft);  vpiImageDestroy(grayRight);
vpiImageDestroy(tempDisparity);
vpiImageDestroy(outDisparity);

/* Destroy CUDA graphics resource */
cudaGraphicsUnmapResources(1, &gres);
cudaGraphicsUnregisterResource(gres);

/* Destroy PBO */
glDeleteBuffers(1, &pbo);
```



FUTURE WORK

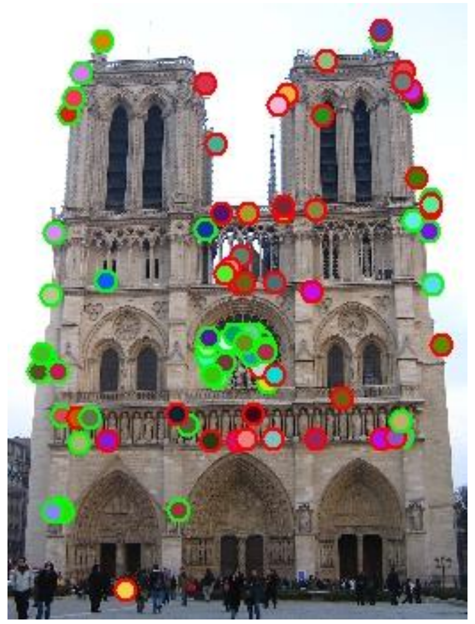
FUTURE WORK

What lies ahead



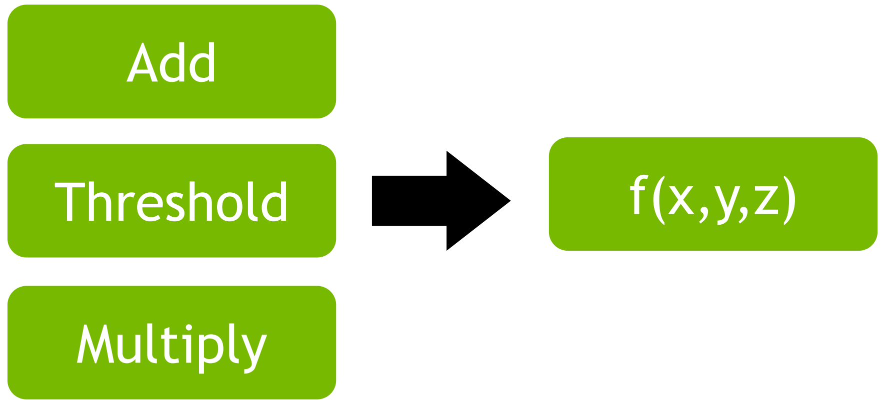
Batching/tensor support for improved DL pre- and post-processing capabilities.

Python bindings.

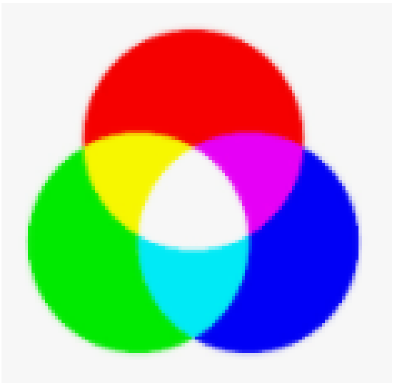


More feature detectors, extractors and matchers.

High-level C++ API.

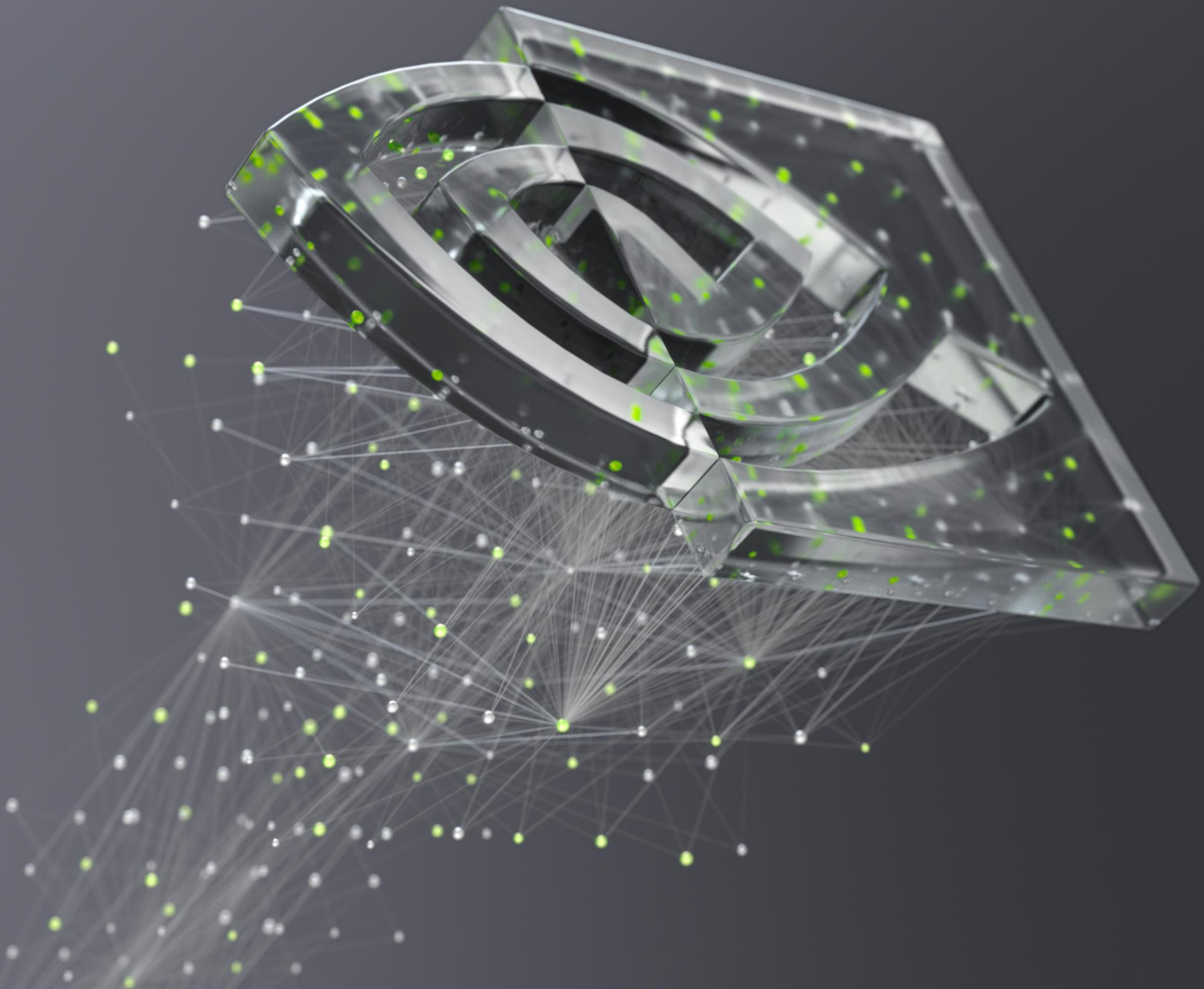


Algorithm fusion, allowing efficient execution of arbitrary pointwise operations, among others.



Accurate end-to-end color handling.

Q & A



Thank You!